Model Definition

A model definition is the most generic type of definition in Modelica. Later in the book (and even in this chapter), we’ll be introducing other types of definitions (*e.g.,* record definitions) that share the same syntax as a model definition, but include some restrictions on what the definition is allowed to contain.

Syntax of a Model Definition

As we saw throughout this chapter, a model definition starts with the model keyword and is followed by a model name (and optionally a model description). The name of the model must start with a letter and can be followed by any collection of letters, numbers or underscores (\_).

**Naming conventions**

Although not strictly required by the language. It is a convention that **model names start with an upper case letter**. Most model developers use the so-called “camel case” convention where the first letter of each word in the model name is upper case.

The model definition can contain variables and equations (to be discussed shortly). The end of the model is indicated by the presence of the end keyword followed by a repetition of the model name. Any text appearing after the sequence // and until the end of the line or between the delimiters /\* and \*/ is considered a comment.

In summary, a model definition has the following general form:

**model** **SomeModelName** "An optional description"

*// By convention, variables are listed at the start*

**equation**

*/\* And equations are listed at the end \*/*

**end** **SomeModelName**;

Inheritance

As we saw in the section on [*Avoiding Repetition*](http://book.xogeny.com/behavior/equations/population/#avoiding-repetition), we can reuse code from other models by adding an extends clause to the model. It is worth noting that a model definition can include multiple extends clauses.

Each extends clause must include the name of the model being extended from and can be optionally followed by modifications that are applied to the contents of the model being extended from. In the case of a model definition that inherits from other model definitions, you can think of the general syntax as looking something like this:

**model** **SpecializedModelName** "An optional description"

**extends** Model1; *// No modifications*

**extends** Model2(n=5); *// Including modification*

*// By convention, variables are listed at the start*

**equation**

*/\* And equations are listed at the end \*/*

**end** **SpecializedModelName**;

By convention, extends clauses are normally listed at the very top of the model definition, before any variables.

In later chapters, we will show how this same syntax can be used to define other entities besides models. But for now, we will focus primarily on models.

# Variables

As we saw in the previous section, a model definition typically contains variable declarations. The basic syntax for a variable declaration is simply the “type” of the variable (which will be discussed shortly in the section on [Built-In Types](http://book.xogeny.com/behavior/equations/variables/#builtin-types)) followed by the name of the variable, e.g.,

Real x;

Variables sharing the same type can be grouped together using the following syntax:

Real x, y;

A declaration can also be followed by a description, e.g.:

Real alpha "angular acceleration";

## Variability

### **Parameters**

By default, variables declared inside a model are assumed to be continuous variables (variables whose solution is generally smooth, but which may also include discontinuities). However, as we first saw in the section titled [Getting Physical](http://book.xogeny.com/behavior/equations/physical/#getting-physical), it is also possible to add the parameter qualifier in front of a variable declaration and to indicate that the variable is known a priori. You can think of a parameter as “input data” to the model that is constant with respect to time.

### **Constants**

Closely related to the parameter qualifier is the constant qualifier. When placed in front of a variable declaration, the constant qualifier also implies that the value of the variable is known a priori and is constant with respect to time. The distinction between the two lies in the fact that a parameter value can be changed from one simulation to the next whereas the value of a constant cannot be changed once the model is compiled. The use of constant by a model developer ensures that end users are not given the option to make changes to the constant. A constant is frequently used to represent physical quantities like π or the Earth’s gravitational acceleration, which can be assumed constant for most engineering simulations.

### **Discrete Variables**

Another qualifier that can be placed in front of a variable declaration is the discrete qualifier. We have not yet shown any example where the discretequalifier would be relevant. However, it is included now for completeness since it is the last remaining variability qualifier.

## Built-In Types

Many of the examples so far referenced the Real type when declaring variables. As the name suggests, Real is used to represent real valued variables (which will generally be translated into floating point representations by a Modelica compiler). However, Real is just one of the four built-in types in Modelica.

Another of the built-in types is the Integer type. This type is used to represent integer values. Integer variables have many uses including representing the size of arrays (this use case will be discussed shortly in an upcoming section on [Vectors and Arrays](http://book.xogeny.com/behavior/arrays/#vectors-and-arrays)).

The remaining built-in types are Boolean (used to represent values that can be either true or false) and String (used for representing character strings).

Each of the built-in types restricts the possible values that a variable can have. Obviously, an Integer variable cannot have the value 2.5, a Boolean orString cannot be 7 and a Real variable cannot have the value "Hello".

## Derived Types

As we saw in the previous examples that introduced [Physical Types](http://book.xogeny.com/behavior/equations/physical/#physical-types), it is possible to “specialize” the built-in types. This feature is used mainly to modify the values associated with [Attributes](http://book.xogeny.com/behavior/equations/variables/#attributes) like unit. The general syntax for creating derived types is:

**type** **NewTypeName** = BaseTypeName(*/\* attributes to be modified \*/*);

Frequently, the BaseTypeName will be one of the built-in types (e.g., Real). But it can also be another derived type. This means that multiple levels of specialization can be supported, e.g.,

**type** **Temperature** = Real(unit="K"); *// Could be a temperature difference*

**type** **AbsoluteTemperature** = Temperature(min=0); *// Must be positive*

## Enumerations

An enumeration type is very similar to the Integer type. An enumeration is typically used to define a type that can take on only a limited set of specific values. In fact, enumerations are not strictly necessary in the language. Their values can always be represented by integers. However, the enumeration type is safer and more readable than an Integer.

There are two built-in enumeration types. The first of these is AssertionLevel and it is defined as follows:

**type** **AssertionLevel** = **enumeration**(warning, error);

The significance of these values will be discussed in a forthcoming section on [assert](http://book.xogeny.com/components/components/component_models/#assertions).

The other built-in enumeration is StateSelect and it is defined as follows:

**type** **StateSelect** = **enumeration**(never, avoid, default, prefer, always);

## Attributes

So far in this chapter we have mentioned attributes (e.g., unit), but we haven’t discussed them in detail. For example, which attributes are present on a given variable? This depends on the type of the variable (and which built-in and derived types it is based on). The following table introduces all the possible attributes indicating their types (i.e., what type of value can be given for that attribute), which types they can be associated with and finally a brief description of the attribute:

### **Attributes of Real**

quantity

A textual description of what the variable represents

**Default**: ""

**Type**: String

start

The start attribute has many uses. The main purpose of the start attribute (as discussed extensively in the section on [Initialization](http://book.xogeny.com/behavior/equations/initialization/#initialization)) is to provide “fallback” initial conditions for state variables (see fixed attribute for more details).

The start attribute may also be used as an initial guess if the variable has been chosen as an iteration variable.

Finally, if a parameter doesn’t have an explicit value specified, the value of the start attribute will be used as the default value for the parameter.

**Default**: 0.0

**Type**: Real

fixed

The fixed attribute changes the way the start attribute is used when the start attribute is used as an initial condition. Normally, the start attribute is considered a “fallback” initial condition and only used if there are insufficient initial conditions explicitly specified in the initial equation sections. However, if the fixed attribute is set to true, then the start attribute is treated as if it was used as an explicit initial equation (i.e., it is no longer used as a fallback, but instead treated as a strict initial condition).

Another, more obscure, use of the fixed attribute is for “computed parameters”. In rare cases where a parameter cannot be initialized explicitly, it is possible to provide a general equation for the parameter in an initial equation section. But in cases where the parameter is initialized in this way, thefixed attribute for the parameter variable must be set to false.

**Default**: false (except for parameter variables, where it is true by default)

**Type**: Real

min

The min attribute is used to specify the minimum allowed value for a variable. This attribute can be used by editors and compilers in various ways to inform users or developers about potentially invalid input data or solutions.

**Default**: -∞

**Type**: Real

max

The max attribute is used to specify the maximum allowed value for a variable. This attribute can be used by editors and compilers in various ways to inform users or developers about potentially invalid input data or solutions.

**Default**: ∞

**Type**: Real

unit

As discussed extensively in this chapter, variables can have physical units associated with them. There are rules about how these units are expressed, but the net result is that by using the unit attribute it is possible check models to make sure that equations are physically consistent. A value of "1"indicates the value has no physical units. On the other hand, a value of "" (the default value if no value is given) indicates that the physical units are simply unspecified. The difference between "1" and "" is that the former is an explicit statement that the quantity is dimensionless (has not units) while the latter indicates that the quantity may have physical units but they are left unspecified.

**Default**: "" (i.e., no physical units specified)

**Type**: String

displayUnit

While the unit attribute describes what physical units should be associated with the value of a variable, the displayUnit expresses a preference for what units should be used when displaying the value of a variable. For example, the SI unit for pressure is Pascals. However, standard atmospheric pressure is 101,325 Pascals. When entering, displaying or plotting pressures it may be more convenient to use bars.

The displayUnit attribute doesn’t affect the value of a variable or the equations used to simulate a model. It only affects the rendering of those values by potentially transforming them into more convenient units for display.

**Default**: ""

**Type**: String

nominal

The nominal attribute is used to specify a nominal value for a variable. This nominal value is generally used in numerical calculations to perform various types of scaling used to avoid round-off or truncation error.

**Default**: 0.0

**Type**: Real

stateSelect

The stateSelect attribute is used as a hint to Modelica compilers about whether a given variable should be chosen as a state (in cases where there is a choice to be made). As discussed previously in the section on [Enumerations](http://book.xogeny.com/behavior/equations/variables/#enumerations), the possible values for this attribute are never, avoid, default, prefer andalways.

**Default**: default

**Type**: StateSelect (enumeration, see [Enumerations](http://book.xogeny.com/behavior/equations/variables/#enumerations))

### **Attributes of Integer**

quantity

A textual description of what the variable represents

**Default**: ""

**Type**: String

start

It is worth noting that an Integer variable can be chosen as a state variable or as an iteration variable. Under these circumstances, the start attribute may be used by a compiler in the same was as it is for Real variables (see previous discussion of [Attributes of Real](http://book.xogeny.com/behavior/equations/variables/#fixed-attribute))

In the case of a parameter, the start attribute will (as usual) be used as the default value for the parameter.

**Default**: 0.0

**Type**: Integer

fixed

see previous discussion of [Attributes of Real](http://book.xogeny.com/behavior/equations/variables/#fixed-attribute)

**Default**: false (except for parameter variables, where it is true by default)

**Type**: Boolean

min

The min attribute is used to specify the minimum allowed value for a variable. This attribute can be used by editors and compilers in various ways to inform users or developers about potentially invalid input data or solutions.

**Default**: -∞

**Type**: Integer

max

The max attribute is used to specify the maximum allowed value for a variable. This attribute can be used by editors and compilers in various ways to inform users or developers about potentially invalid input data or solutions.

**Default**: ∞

**Type**: Integer

### **Attributes of Boolean**

quantity

A textual description of what the variable represents

**Default**: ""

**Type**: String

start

It is worth noting that an Boolean variable can be chosen as a state variable or as an iteration variable. Under these circumstances, the start attribute may be used by a compiler in the same was as it is for Real variables (see previous discussion of [Attributes of Real](http://book.xogeny.com/behavior/equations/variables/#fixed-attribute))

In the case of a parameter, the start attribute will (as usual) be used as the default value for the parameter.

**Default**: 0.0

**Type**: Boolean

fixed

see previous discussion of [Attributes of Real](http://book.xogeny.com/behavior/equations/variables/#fixed-attribute)

**Default**: false (except for parameter variables, where it is true by default)

**Type**: Boolean

### **Attributes of String**

quantity

A textual description of what the variable represents

**Default**: ""

**Type**: String

start

Technically, a String could be chosen as a state variable (or even an iteration variable), but in practice this never happens. So for a String variable the only practical use of the start attribute is to define the value of a parameter (that happens to have the type of String) if no explicit value for the parameter is given.

**Default**: ""

**Type**: String

It is worth noting that [Derived Types](http://book.xogeny.com/behavior/equations/variables/#derived-types) retain the attributes of the built-in type that they are ultimately derived from. Also, although the type of, for example, the min attribute on a Real variable is listed having the type Real it should be pointed out explicitly that attributes cannot themselves have attributes. In other words, the start attribute doesn’t have a start attribute.

## Modifications

So far, we’ve seen two types of modifications. The first is when we change the value of an attribute, e.g.,

Real x(start=10);

In this case, we are creating a variable x of type Real. But rather than leaving it “as is”, we then apply a modification to x. Specifically, we “reach inside” of xand change the start attribute value. In this example, we are only going one level into x to make our modification. But as we will see in our next example, it is possible to make modifications at arbitrary depths.

The other case where we have seen modifications was in the section on [Avoiding Repetition](http://book.xogeny.com/behavior/equations/population/#avoiding-repetition). There we saw modification used in conjunction with extendsclauses, e.g.,

**extends** QuiescentModelWithInheritance(gamma=0.3, delta=0.01);

Here, the modification is applied to elements that were inherited from the QuiescentModelWithInheritance model. As with modifications to attributes, the element being modified (a model in this case) is followed by parentheses and inside those parentheses we specify the modifications we wish to make.

It is worth noting that modifications can be nested arbitrarily deep. For example, imagine we wanted to modify the start attribute for the variable xinherited from the QuiescentModelWithInheritance model. In Modelica, such a modification would be made as follows:

**extends** QuiescentModelWithInheritance(x(start=5));

Here we first “reach inside” the QuiescentModelWithInheritance model to modify the contents that we “inherit” from it (x in this case) and then we “reach inside” x to modify the value of the start attribute.

One of the central themes of Modelica is support for reuse and avoiding the need to “copy and paste” code. Modifications are one of the essential features in Modelica that support reuse. We’ll learn about others in future sections.

# Equations

Although equations are probably the single most important mathematical aspect of Modelica, they are also the simplest to explain.

## Basic Equations

There are really no complicated semantics to explain about equations. All equations are composed of a left hand expression and a right hand expression separated by an equals sign, i.e.,

<left-hand expression> = <right-hand expression>;

Through the examples presented in this chapter, the reader has been exposed to this pattern over and over again in each example. The only real deviation from the syntax shown above is the case where a description of the equation is included as well, e.g.,

V = i\*R "Ohm's law";

m\*der(v) = F "Newton's law";

As was pointed out previously, the left hand and right hand sides of an equation in Modelica are expressions, not assignments. In other words (and in contrast to most programming languages), the left hand side does **not** have to be a variable (as we can see in the case of Newton’s law above).

## Initial Equations

As we saw in many of the examples in this chapter, it is possible to specify equations within a model to be used to solve for initial conditions. This entire topic of initialization will be discussed in detail in the next section, titled [Initialization](http://book.xogeny.com/behavior/equations/initialization/#initialization). For now, all we will say on this topic is that if an equation is to be applied only to solve for initial conditions, the equation section must be qualified by the initial keyword as follows:

**initial equation**

x = 0; *// Only used to solve for initial conditions*

## Conditional Equations

In the next chapter, we’ll discuss how to use if statements to represent conditional behavior. It is worth getting ahead of ourselves a little bit to point out that equations can be conditional. There are really two forms of conditional equations. The first is the balanced form, e.g.,

**if** a>b **then**

x = 5\***time**;

**else**

x = 3\***time**;

**end** **if**;

In the balanced case, the number of equations is always the same (1 in the code above), but which equation can change. This is important because to simulate a model in Modelica, the number of variables must equal the number of equations and the number of equations must be fixed during the simulation.

The other type of conditional equations are ones where the number of equations is unbalanced. This means that the number of equations on the if side may not be equal to the number of equations on the else side (like it was in the balanced case, previously).

But remember, the number of equations cannot change during a simulation. So how is it then that the number of equations can be different from the if side to the else side? It can only happen if **the value of the conditional expression cannot change during the simulation**. In order to be able to ensure that the conditional expression can never change, it is necessary that all variables in the conditional expression have so-called parametric variability.

Remember in our discussion of [Variability](http://book.xogeny.com/behavior/equations/variables/#variability) the fact that variables with the parameter qualifier cannot change during a simulation? If a variable with theparameter qualifier cannot change during a simulation and all the variables in an expression have this parametric variability then the entire expression must also have parametric variability (i.e., the value of the expression cannot change during a simulation).

At this point, you might be asking yourself why this unbalanced case would be useful? Again, we are getting ahead of ourselves here, but one use case would be the conditional application of initial equations, e.g.,

..

**parameter** Boolean steady\_state;

**initial equation**

**if** steady\_state **then**

der(x) = 0;

der(y) = 0;

..

In other words, if the Boolean parameter steady\_state is true, then the initial equations are enforced. But if the parameter is false, they are not. The conditional expression here clearly has parametric variability because the expression contains only a variable and that variable is a parameter.

That’s all we’ll say on this topic for now, since discrete and conditional behavior will be discussed in detail in the [next chapter](http://book.xogeny.com/behavior/discrete/#discrete-behavior).

Initialization

Overview

As we already touched on during our previous discussion on [*Steady State Initialization*](http://book.xogeny.com/behavior/equations/population/#steady-state), behavior is represented by both the equations contained in a model as well as the initial conditions given to the state variables in the model. In Modelica, the initial conditions are computed by combining the normal equations (present in *equation* sections) with any initial equations (present in *initial equation* sections).

One of the first sources of confusion for new users is understanding how many initial conditions are required. The answer to this question is simple. In order to have a well-posed initialization problem (one where we don’t have too many or too few initial equations), we need to have the same number of equations in the *initial equation* sections as we have states in our system. **Note**, we can get away with having too few, because tools can augment the initial equations we provide with additional ones until the problem is well-posed, but we cannot solve a problem where we have too many initial equations.

Of course, saying the number of initial equations has to be equal to the number of states answers one question, but quickly creates another, *i.e.,* *how do we determine how many states there are*? For the models we’ve seen in this chapter, the answer is quite simple. The states in each of our examples so far are the variables that appear inside the der(...) operator. In other words, every variable that we differentiated in those examples is a state.

Ordinary Differential Equations

It is important to note that **it will not always be the case** that every variable that we differentiate will be a state. In this chapter, all the models we have seen so far are ordinary differential equations (ODEs). When dealing with ODEs, every differentiated variable is a state, which, in turn, means that you need an initial equation for each of these differentiated variables. But in subsequent chapters we will eventually run across examples that are so-called differential-algebraic equations (DAEs). In those cases, only *some* of the differentiated variables can be considered states.

As it turns out, understanding initialization doesn’t really require us to get into a detailed discussion about DAEs. In practice, all Modelica tools perform something called “index reduction”. While the index reduction algorithms themselves are fairly complicated (so we won’t get into those now), the effect is quite simple. Index reduction transforms the DAEs into ODEs. In other words, Modelica compilers will transform whatever DAE problem contained in our Modelica code into this relatively easy to explain ODE form.

So let’s side-step the discussion about DAEs and index reduction and just pick up our discussion of initialization assuming our problem has already been reduced to an ODE. In this case, the only thing we really need to understand is that initialization is required for all states in the model and that our model will have the following general ODE form:

x⃗ ˙(t)= f⃗ (x⃗ (t),u⃗ (t),t)

y⃗ (t)=g⃗ (x⃗ (t),u⃗ (t),t)

where t is the current simulation time, x⃗ (t) are the values of the states in our system at time t, u⃗ (t) are the values of any external inputs to our system at time t.

Note that the arrow over a variable simply indicates that it is a vector, not a scalar. Also note that the only variable that appears differentiated in this problem is x⃗ . This is how we know that x⃗  represents the states in the system. One final thing to note about this system is that neither function, f⃗  nor g⃗ , depends on y⃗ .

If you think about it, both t and u⃗ (t) are external to our system. We don’t compute them or control them. The reason that we call x⃗  the state of our system is that it the only information (from within our system) needed to compute x⃗ ˙(t) and y⃗ (t) (which, in turn, are the only things we need to compute in order to arrive at a solution).

Getting back to the topic of initialization, during a normal time step we will solve for x⃗ (t) by integrating x⃗ ˙(t) to compute x⃗ (t). In other words:

x⃗ (T)=∫Ttix⃗ ˙(t) dt+x⃗ (ti)

This all works as long as there **was** a previous time step. When there wasn’t a previous time step, then the value of x⃗  that we plug into our equations has to be the very first value of x⃗  in our simulation. In other words, our initial conditions.

One might imagine that we would specify our initial conditions by adding an equation like this:

x⃗ (t0)=x⃗ 0

where t0 is the start time of our simulation and x⃗ 0 is an explicit specification of the initial values. Providing explicit values for states is a very common case when specifying initial conditions. So we definitely need to be able to handle this case. But this approach won’t work for the cases we showed in [*Steady State Initialization*](http://book.xogeny.com/behavior/equations/population/#steady-state). There we didn’t provide explicit initial values for states. Instead, we provided initial values for x⃗ ˙(t0). So how can we capture both of these cases?

Initial Equations

The answer is to assume that at the start of our simulation we need to solve a problem that looks like this:

x⃗ ˙(t0) =f⃗ (x⃗ (t0),u⃗ (t0),t0

y⃗ (t0) )=g⃗ (x⃗ (t0),u⃗ (t0),t0)

0⃗ =h⃗ (x⃗ (t0),x⃗ ˙(t0),u⃗ (t0),t0)

Note the introduction of a new function, h⃗ . This new function represents any equations we have placed in *initial equation* sections. The fact that h⃗  takes both x⃗  **and** x⃗ ˙ as arguments allows us to express a wide range of initial conditions. To define explicit initial values for states, we could define h⃗  as:

h⃗ (x⃗ (t0),x⃗ ˙(t0),u⃗ (t0),t0)=x⃗ (t0)−x⃗ 0

But we could also express our desire to start with a steady state solution by defining h⃗  as:

h⃗ (x⃗ (t0),x⃗ ˙(t0),u⃗ (t0),t0)=x⃗ ˙(t0)

And, of course, we could mix these different forms or use a wide range of other forms on a per state basis to describe our initial conditions. So when writing initial equations, all you need to keep in mind is that they need to be of the general form shown above and that you cannot have more of them than you have states in your system.

Conclusion

As we’ve demonstrated in this chapter, the *initial equation* construct in Modelica allows us to express many ways to initialize our system. In the end, all of them will compute the initial values for the states in our system. But we are given tremendous latitude in describing exactly how those values will be computed.

This is an area where Modelica excels. Initialization is given first class treatment in Modelica and this flexibility pays off in many real world applications.

# Record Definitions

Earlier, we introduced the idea of a model definition. Although we haven’t seen any yet, Modelica also includes a record type. A record can have variables, just like a model, but it is not allowed to include equations. Records are primarily used to group data together. But as we will see shortly, they are also very useful in describing the data associated with [Annotations](http://book.xogeny.com/behavior/equations/annotations/#annotations).

## Syntax

The record definition looks essentially like a model definition, but without any equations:

**record** **RecordName** "Description of the record"

*// Declarations for record variables*

**end** **RecordName**;

As with a model, the definition starts and ends with the name of the record being defined. An explanation of the record can be included as a string after the name. All the variables associated with the record are declared within the record definition.

The following are all examples of record definitions:

**record** **Vector** "A vector in 3D space"

Real x;

Real y;

Real z;

**end** **Vector**;

**record** **Complex** "Representation of a complex number"

Real re "Real component";

Real im "Imaginary component";

**end** **Complex**;

## Record Constructors

Now that we know how to define a record, we need to know how to create one. If we are declaring a variable that happens to be a record, the declaration itself will create an instance of the record and we can specify the values of variables inside the record using modifications, e.g.,

**parameter** Vector v(x=1.0, y=2.0, z=0.0);

But there are some cases where we might want to create an instance of a record that isn’t a variable (e.g., to use in an expression, pass as an argument to a function or use in a modification). For each record definition, a function is automatically generated with the **same name** as the record. This function is called the “record constructor”. The record constructor has input arguments that match the variables inside the record definition and returns an instance of that record. So in the case of the Vector definition above, we could also initialize a parameter using the record constructor as follows:

**parameter** Vector v = Vector(x=1.0, y=2.0, z=0.0);

In this case, the value for v comes from the **expression** Vector(x=1.0, y=2.0, z=0.0) which is a call to the record constructor.

# Annotations

Recall in the discussion on [Experimental Conditions](http://book.xogeny.com/behavior/equations/first_order/#experimental-conditions) we included information about the simulation start and stop time using an annotation. An annotationis a way to include information that is not related to the behavior of the model. In the case of experimental conditions, we injected information about how a particular model should be simulated. But annotations are used extensively in Modelica to provide all kinds of additional information about models. For example, as we’ll see [later in the book](http://book.xogeny.com/components/connectors/annotations/#graphical-annos), annotations are used to describe the graphical appearance of components and connectors. For now, the important thing is to understand that annotations are additional data, above and beyond behavior, that can be “attached” to different elements in Modelica.

In this section, we will first cover where an annotation can appear in a Modelica model. Next, we’ll explain how we can use [Record Definitions](http://book.xogeny.com/behavior/equations/record_def/#record-def) to describe the contents of an annotation. Finally, we’ll describe a few of the many “standard” annotations that are included as part of the Modelica specification.

## Annotation Locations

Annotations can appear in many different places in Modelica. We will discuss each potential location and demonstrate the syntax for each case.

### **Declaration Annotations**

A declaration annotation comes at the end of a declaration, right before the ;. Here is a simple declaration that includes an annotation:

**parameter** Real length "Rod length" **annotation**(...);

Note that the annotation comes after the descriptive string and before the ;. Also, the ... is simply a place holder for the [Annotation Data](http://book.xogeny.com/behavior/equations/annotations/#annotation-data), which will be discussed shortly.

### **Statement and Equation Annotations**

It is also possible to associate annotations with equations, for example:

T = T0 "Specify initial value for T" **annotation**(...);

In declarations and equations, the annotation is always at the very end and comes immediately before the ;.

### **Inheritance Annotations**

We briefly discussed the extends keyword when we talked about [Modifications](http://book.xogeny.com/behavior/equations/variables/#modifications) and [Avoiding Repetition](http://book.xogeny.com/behavior/equations/population/#avoiding-repetition). It is possible to associate an annotation with anextends clause as follows:

**extends** QuiescentModelWithInheritance(gamma=0.3, delta=0.01) **annotation**(...);

As we’ve observed in each previous case, the annotation immediately precedes the ;.

### **Model Annotations**

A model annotation associates annotation data directly with the model definition itself. This is exactly the kind of annotation we saw when describing[Experimental Conditions](http://book.xogeny.com/behavior/equations/first_order/#experimental-conditions), e.g.,

**model** **FirstOrderExperiment** "Defining experimental conditions"

Real x "State variable";

**initial equation**

x = 2 "Used before simulation to compute initial values";

**equation**

der(x) = 1-x "Drives value of x toward 1.0";

**annotation**(experiment(StartTime=0,StopTime=8));

**end** **FirstOrderExperiment**;

Note how, unlike all the previous annotation locations we’ve described, this annotation isn’t really “attached” to anything. This indicates that it is annotating the model itself.

## Annotation Data

### **General Syntax**

The syntax of an annotation is the same syntax used for [Modifications](http://book.xogeny.com/behavior/equations/variables/#modifications). This means the annotation will include either an assignment to a variable in the annotation, e.g.,

**annotation**(Evaluate=**true**);

or it will include a modification to something **inside** a variable in the annotation, e.g.,

**annotation**(experiment(StartTime=0,StopTime=8));

### **User Annotations**

Annotations were designed to allow model developers to attach **arbitrary data** to their models. For example, if a user wanted to associate a part number with a given model definition, they might introduce a model annotation like this:

**annotation**(PartNumber="FF78-E4B879");

A general principle of annotation data is that if a tool reads in a model, **it must preserve the annotation information** when it writes it back out. The tool does not (and, in general, will not) have to understand the data. But the data must be preserved.

### **Multiple Annotations**

Imagine a user wanted to specify **both** a part number and an experiment annotation. Then they might end up with an annotation like this one:

**annotation**(PartNumber="FF78-E4B879",

experiment(StartTime=0,StopTime=8));

Note how these two pieces of information can exist side by side. One way to think about annotations is to visualize them as a tree like this:

* PartNumber="FF78-E4B879"
* experiment
  + StartTime=0
  + StopTime=8

### **Namespaces**

This introduces another principle of annotations which is that it should be possible to have more than one **as long as the names are different**. For this reason, choosing names is very important and they should be chosen to avoid potential conflicts with other names. For example, a better approach for including the part number would be to enclose it in a variable that is more likely to be unique to your company or application, e.g.,:

**annotation**(XogenyIndustries(PartNumber="FF78-E4B879"),

experiment(StartTime=0,StopTime=8));

In this case, the variable XogenyIndustries can be used to carve out a “namespace” for a specific organization or purpose. If another organization came along and wanted to associate a different part number with the same model, they could do that by establishing their own separate hierarchy in the annotation, e.g.,:

**annotation**(XogenyIndustries(PartNumber="FF78-E4B879"),

AcmeEquipment(PartNumber="A23335-992"),

experiment(StartTime=0,StopTime=8));

Occasionally, Modelica tool vendors include their own special annotations (e.g., in the Modelica Standard Library). By convention, tool vendors use names that are prefixed by two underscores, e.g.,

**annotation**(XogenyIndustries(PartNumber="FF78-E4B879"),

\_\_ModelicateTechnologies(enableCoolFeature10=**true**),

AcmeEquipment(PartNumber="A23335-992"),

experiment(StartTime=0,StopTime=8));

### **Intepretation**

Remember that annotation data is arbitrary. This allows arbitrary data to be associated with the model. The **meaning** of that data is, in general, not defined in the Modelica specification. As we will see shortly, there are a few “standard” annotations (they will be described throughout this book) and they are documented in the specification. But when users add annotations beyond the standard annotations it is assumed that they have some way (using some Modelica tool, compiler or other Modelica aware technology) of extracting and interpreting their annotation data.

The bottom line is that while you can inject (non-standard) annotation data into the model, tools are only required to preserve it and not to interpret it.

### **Documentation**

It is very common to document Modelica annotations **as if** they had [Record Definitions](http://book.xogeny.com/behavior/equations/record_def/#record-def) associated with them. We’ll see several examples of this technique in our next topic. Using this approach to document expected annotation data are strongly encouraged. In fact, this technique is so popular and useful that there are proposals to actually make it part of the language itself in the future.

## Introductory Annotations

This section introduces just a few of the “standard annotations” in Modelica. As discussed previously, annotations are generally allowed to include arbitrary data that is preserved by tools and, presumably, interpreted at some point. The syntax and meaning of the standard annotations are described in the Modelica specification so they can be interpreted consistently and universally by Modelica tools.

We will follow a convention (whenever possible) of describing standard annotations in terms of record definitions. These record definitions don’t formally exist, they are simply a concise way of expressing the data contained in the annotation.

### **Documentation**

**Type: Model Annotation**

The Documentation annotation in Modelica allows raw text or HTML to be associated with a model as documentation. This documentation is composed of two components. The first is information about the model and the second is revision history information. The structure of the Documentation annotation is described by the following record definition:

**record** **Documentation**

String info "Documentation in text or HTML format";

String revision "Revision information in text or HTML format";

**end** **Documentation**;

When embedding HTML inside an annotation, the HTML code must be surrounded by <html> tags, e.g.,

**model** **MyWidget**

*// ... declarations*

**annotation**(

Documentation(

info=**"<html><h1** class=\"heading\"**>**Introduction**</h1><p>**...**</p></html>"**));

*// .. equations*

**end** **MyWidget**;

Here the model MyWidget contains HTML documentation. The documentation is wrapped by <html> tags **and all quotes used to define attributes are escaped by \”** to avoid accidentally terminating the info string.

### **experiment**

**Type: Model Annotation**

The experiment annotation is used to specify information about how a given model should be simulated. The annotation data can be represented in recordform as:

**record** **experiment**

Real StartTime "Time at which the simulation should start";

Real StopTime "Time at which the simulation should stop";

Real Interval(min=0) "Time interval between results";

Real Tolerance(min=0) "Solver tolerance to use";

**end** **experiment**;

### **Evaluate**

**Type: Declaration Annotation (applies to parameters)**

The Evaluate annotation indicates to a Modelica compiler that the value of a given parameter can be transformed into a constant at compile time. In other words, it indicates that the user does not require the ability to change the value of the parameter from one simulation to the next.

The motivation behind having such an annotation is that it allows the Modelica compiler to assume many things about the parameter during model compilation that it otherwise couldn’t. These assumptions might restrict the system of equations in such a way that the underlying systems of equations are easier to solve than in the general case where the parameter could take on a range of values.

The Evaluate annotation is simply a Boolean variable (true indicating that the parameter value can be transformed into a constant). It is used in an annotation as follows:

**parameter** Real x **annotation**(Evaluate=**true**);

### **HideResult**

**Type: Declaration Annotation**

The HideResult annotation is used to indicate that the solution for a given variable is not of interest to the analyst. By setting the value of HideResult to true, the model developer is indicating to the Modelica compiler that it need not store the annotated variable in any simulation results that are produced. This can save both simulation time and disk space because it avoids writing out data that will never be viewed.

The HideResult annotation would be used as follows:

Real z "Uninteresting variable" **annotation**(HideResult=**true**);