CHAPTER 10 HOW (AND WHY) TO WRITE STANDARD METHODS

In this chapter we describe a methodology (pun intended) which can help make anyone who can solve a quadratic equation a mathematical modeling expert. This methodology helps you to avoid mistakes and to find mistakes quickly when you make them. Finding bugs weeks after creating a model is annoying, inefficient, and (frequently) embarrassing. Because METHOD code can be large, we do not include many examples here. See Chapter 3, “Preparing a model for reuse,” on page 25 for detailed examples. One of the advantages of this methodology is that it allows almost automatic generation of methods for a model based on the declarative structure (defined parts and variables) in the model, as we shall see in Section 10.10. Even if you skip much of this chapter, read Section 10.10

We divide methods into _self and _all categories. The premise of this method design is that we can write the _self methods incrementally, building on the already tested methods of previous MODEL parts we are reusing. In this way we never have to write a single huge method that directly manipulates 100s of variables in a hierarchy. The _all methods are methods which simply “top off” the _self methods. With an _all method, you can treat just a *part* of a larger simulation already built as a self-contained simulation.

Usually discovery of the information you need to write the methods proceeds in the order that they appear below: check, default, specify, bound, scale.

10.1 WHY YOU SHOULD FOLLOW OUR WAYS

If debugging is the repair of modeling errors, then modeling must be the process of creating those errors.[1]

Some geniuses make more mistakes than anyone else -- because they try more things that anyone else. Part (perhaps a very large part) of what makes such a genius different from the rest of humanity is that they quickly recognize their own mistakes and move on to something else before anyone notices that they have screwed up! Solving a problem as far and as fast as you can, and then going back to criticize

1. Somebody famous said something like this about programming computers. The principle holds.

every aspect of the solution with an eye to improving it is how you usually discover right answers. Do it our way so that **ASCEND can help you find your mistakes.**

We (geniuses or not we'll leave to our users to decide) have found that it is best to write mathematical MODELs and mathematical modeling software in ways which make our mistakes (or your mistakes) very easy to detect. This way it is easier to find and fix problems early, instead of discovering the bug while the boss and the vice-president (or the advisor and the industrial sponsor) are hovering near. The ASCEND system will not force you to write standard methods in your models. METHODs of the sort we advocate here make your MODELs much easier to use and much more reliable. They pay off in the short run as well as the long run. These are *guidelines*, not *laws*: geniuses know when to color outside the lines.

If you do not write the standard methods, your MODEL will inherit the ones given in the library basemodel.a4l. The *ClearAll* and *reset* methods here will work for you if you follow the guidelines for the method *specify*. The other methods defined in basemodel.a4l (*check_self, default_self, bound_self, scale_self, check_all, default_all, bound_all, scale_all*) all contain STOP statements which will warn you that you have skipped something important, should you accidentally call one of these methods. If you create a model for someone else and they run into one of these STOP errors while using your model, that error is *your* fault.

10.2 METHODS *_SELF VS *_ALL

When you create a model definition, you create a container holding variables, equations, arrays, and other models. You create methods in the same definition to control the state of (the values stored in) all these parts. ASCEND lets you share objects among several models by passing objects through a model interface (the MODEL parameter list), by creating ALIASES for parts within contained objects, and even by merging parts (though this is a dumb idea for any object bigger than a variable).

Too many cooks spoil the soup.

The problem this creates for you as a METHOD writer is to decide which of the several MODELs that share an object is responsible for updating that variable's default, bounds, and nominal values. You could decide that every model which shares a variable is responsible for these values. This will lead to many, many, many hard to understand conflicts as different models all try to manage the same value. The sensible approach is to make only one model responsible for the bounding,

scaling, and default setting of each variable: the model which creates the variable in the first place.

Use *_self methods on locally created variables and parts

Consider the following model and creating the *_self methods default_self, check_self, bound_self, and scale_self for it.

```
MODEL selfish(
   external_var WILL_BE solver_var;
   out_thingy WILL_BE input_part;
);
   my_variable IS_A solver_var;
   peek_at_variable ALIASES out_thingy.mabob.cost;
   my_thingy IS_A nother_part;
   navel_gaze ALIASES my_thingy.mabob.cost;
END selfish;
```

This model should manage the value of the variable it creates: *my_variable*. *External_var* comes in from the outside, so some other model will create and manage it. *Peek_at_variable* and *navel_gaze* also are not created here and should not be managed in the *_self methods of *selfish*.. We want to default, bound, or scale variables in complex parts we create also. We should call *my_thingy.default_self* whenever *default_self* is called for this model. We should not call *out_thingy.default_self*, however, as some other model will do so.

Use *_all methods to manage a troublesome part

Any mathematical subproblem in a large simulation may need to be isolated for debugging or solving purposes. When this is done using the Browser and Solver tools, you still need to call scaling, bounding, and checking methods for all parts of the isolated subproblem, even for those parts that came in from the outside. This is easily done by writing *_all methods. In the example above, *scale_all* will scale *external_var* and call *out_thingy.scale_all* because these parts are defined using WILL_BE statements. *scale_all* will then call the local*scale_self* to do all the normal scaling.

That’s the big picture of _self and _all methods. Each kind of method (bound, scale, default, check) has its own peculiarities which we cover in Section 10.4 and Section 10.5, but they all follow the rules above which distinguish among variables and parts defined with WILL_BE (managed in *_all only), IS_A (managed in *_self only), and ALIASES (not our responsibility).

10.3 HOW TO WRITE CLEARALL AND RESET

Writing these two standard methods in your model is very simple— do nothing. You may wish to write alternative reset_* methods as we shall discuss. These methods are inheritted by all models from the definitions in basemodel.a4l. Just so you know, here is what they do.

10.3.1 CLEARALL

This method finds any variable that is a solver_var or refinement of solver_var and changes the *.fixed* flag on that var to FALSE. This method does not change the value of *.included* flags on relations or return boolean, integer, or symbol variables to a default value.

10.3.2 RESET

This method calls *ClearAll* to bring the model to a standard state with all variables unfixed (free), then it calls the user-written *specify* method to bring the model with an equal number of variables to calculate and equations to solve. Normally you do not need to write this method: your models will inherit this one unless you override it (redefine it) in your MODEL.

This standard state is not necessarily the most useful starting state for any particular application. This method merely establishes a base case. There is no ‘one perfect “reset”’ for all purposes. Other reset_whatElseYouWant methods can also be written. The name of a method is a communication tool. Please use meaningful names as long as necessary to tell what the method does. Avoid cryptic abbreviations and hyper-specialized jargon known only to you and your three friends when you are naming methods; however, do not shy away from technical terms common to the engineering domain in which you are modeling.

10.4 THE *_SELF METHODS

The following methods should be redefined by each reusable library MODEL. Models that do not supply proper versions of these methods are usually very hard to reuse.

10.4.1 METHOD CHECK_SELF

This method should be written first, though it is run last. Just like they taught you in elementary school, always check your work. Start by defining criteria for a successful solution that will not be included in the equations solved and compute them in this method. As you develop your MODEL, you should expect to revise the check method from time to time, if you are learning anything about the MODEL. We frequently change our definition of success when modeling.

When a mathematical MODEL is solved, the assumptions that went into writing (deriving) the equations should be checked. Usually there are redundant equations available (more than one way to state the physics or economics mathematically). These should be used to check the particularly tricky bits of the MODEL.

Check that the physical or intuitive (qualitative) relationships among variables you expect to hold are TRUE, especially if you have not written such relationships in terms of inequalities (x*z <= y) in the MODEL equations.

In some models, checking the variable values against absolute physical limits (temperature > 0{K} and temperature < Tcritical for example) may be all that is necessary or possible. Do not check variable values against their .lower_bound or .upper_bound, as any decent algebraic solver or modeling system (e.g. ASCEND) will do this for you.

If a check fails, use a STOP statement to notify yourself (or you MODEL using customer) that the solution may be bogus. STOP raises an error signal and issues an error message. STOP normally also stops further execution of the method and returns control to a higher level, though there are interactive tools to force method execution to continue. STOP does not crash the ASCEND system.

10.4.2 METHOD DEFAULT_SELF

This method should set default values for any variables declared locally (IS_A) to the MODEL. It also should run *default_self* on *all* the complex parts that are declared locally (with IS_A) in the MODEL. If the atoms you use to define your variables have a suitable default already, then you do not need to assign them a default in this method.

This method should not run any methods on MODEL parts that come via WILL_BE in the definition’s parameter list. This method also should not change the values of variables that are passed in through the

parameter list. Sometimes there will be nothing for this method to do. Define it anyway, leaving it empty, so that any writer reusing this model as part of a higher level model can safely assume it is there and call it without having to know the details.

When a top-level simulation is built by the compiler, this method will be run (for the top-level model) at the end of compilation. If your model's *default_self* method does not call the lower level *default_self* methods in your model's IS_A'd parts, it is quite likely that your model will not solve.

10.4.3 METHOD BOUND_SELF

Much of the art of nonlinear physical modeling is in bounding the solution. This method should update the bounds on locally defined (IS_A) variables and IS_A defined MODEL parts. Updating bounds requires some care. For example, the bounds on fractions frequently don't need updating. This method should not bound variables passed into the MODEL definition or parts passed into the definition.

A common formula for updating bounds is to define a region around the current value of the variable. A linear region size formula, as an example, would be:

$$\text{x.bound} = x \pm \Delta \bullet \text{x.nominal}$$

or

```
v.upper_bound := v + boundwidth * v.nominal;
v.lower_bound := v - boundwidth v.nominal;
```

Care must be taken that such a formula does not move the bounds (particularly lower bounds) out so far as to allow non-physical solutions. Logarithmic bounding regions are also simple to calculate. Here boundwidth IS_A bound_width; boundwidth is a real atom (but not a solver_var) or some value you can use to determine how much "wiggle-room" you want to give a solver.

Small powers of 4 and 10 are usually good values of boundwidth. Too small a boundwidth can cut off the portion of number space where the solution is found. Too large a bound width can allow solvers to wander for great distances in uninteresting regions of the number space.

10.4.4 METHOD SCALE_SELF

Most nonlinear (and many linear) models cannot be solved without proper scaling of the variables. *scale_self* should reset the .nominal value on every real variable in need of scaling. It should then call the *scale_self* method on all the locally defined (IS_A) parts of the MODEL. 0.0 is the worst possible nominal value. A proper nominal is one such that you expect at the solution $0.1 \leq abs(\frac{x}{x.nominal}) \leq 10$. This method should not change the scaling of models and variables that are received through the parameter list of the MODEL.

Variables (like fractions) bounded such that they cannot be too far away from 1.0 in magnitude probably don't need scaling most of the time if they are also bounded away from 0.0.

Some solvers, but not all, will attempt to scale the equations and variables by heuristic matrix-based methods. This works, but inconsistently; user-defined scaling is generally superior. ASCEND makes scaling equations easy to do. You scale the variables, which can only be done well by knowing something about where the solution is going to be found (by being an engineer, for example.) Then ASCEND can calculate an appropriate equation-scaling by efficient symbolic methods.

10.5 THE *_ALL METHODS

10.5.1 METHOD DEFAULT_ALL

This method assumes that the arguments to the MODEL instance have not been properly initialized, as is frequently the case in one-off modeling efforts. This method should run the *default_all* method on each of the parts received through the parameter list via WILL_BE statements and should give appropriate default values to any variables received through the parameter list. After these have been done, it should then call *default_self* to take care of all local defaults.

10.5.2 METHOD CHECK_ALL

When solving only a part of a simulation, it is necessary to check the models and variables passed into the part as well as the locally defined parts and variables. This method should call *check_all* on the parts received as WILL_BE parameters, then call *check_self* to check the locally defined parts and equations.

10.5.3 METHOD BOUND_ALL

This method should be like *bound_self* except that it bounds the passed in variables and calls *bound_all* on the passed in parts. It should then call *bound_self*.

10.5.4 METHOD SCALE_ALL

This method should be like *scale_self* above except that it scales the variables received through the parameter list and calls scale_all on the passed in parts. It should then call *scale_self* to take care of the local variables and models.

10.6 METHOD SPECIFY

Assuming ClearAll has been run on the MODEL, this method should get the MODEL to a condition called ‘square’: the case where there are as many variables with .fixed == FALSE as there equations eligible to compute them. This is one of the hardest tasks ever invented by mathematicians if you go about it in the wrong way. We think we know the right way.

Actually, ‘square’ is a bit trickier to achieve than simply counting equations and variables. Solvers, such as QRSlv in ASCEND, may help greatly with the bookkeeping.

The general approach is to:

1. Run “specify” for all the parts (both passed in and locally defined) that are not passed on into other parts.
2. Fix up (by tweaking .fixed flags on variables) any difficulties that arise when parts compete to calculate the same variable.
3. Use the remaining new local variables to take care of any leftover equations among the parts and any new equations written locally.

At all steps 1-3 pay special attention to indexed variables used in indexed equations. Frequently you must fix or free N or N-1 variables indexed over a set of size N, if there are N matching equations. In general, if you think you have *specify* correctly written, change the sizes of all the sets in your MODEL by one and then by two members. If your specify method still works, you are probably using sets correctly. Pursuing “symmetry,” the identical treatment of all variables defined in a single array, usually helps you write specify correctly.

When writing models that combine parts which do not share very well, or which both try to compute the same variable in different ways, it may even be necessary to write a WHEN statement to selectively TURN OFF the conflicting equations or MODEL fragments. An object or equation USEd in any WHEN statement is turned off by default and becomes a part of the solved MODEL only when the condition of some CASE which USEs that object is matched.

The setting of boolean, integer, and symbol variables which are controlling conditions of WHEN and SWITCH statements should be done in the specify method.

There is no ‘one perfect “specify”’ for all purposes. This routine should merely define a reasonably useful base configuration of the MODEL. Other specify_whatElseYouWant methods can also be written. Again, the name of a method is a communication tool. Please use meaningful names as long as necessary to tell what the method does. Avoid cryptic abbreviations and hyper-specialized jargon known only to you and your three friends when you are naming methods; however, do not shy away from technical terms common to the engineering domain in which you are modeling.

10.7 METHOD VALUES

In a final application MODEL, you should record at least one set of input values (values of the fixed variables and guesses of key solved-for variables) that leads to a good solution. Do this so no one need reinvent that set the next time you use the MODEL or someone picks the MODEL up after you.

10.8 METHODS AND CHEMICAL PROCESS MODELS

This next tip is due to Duncan Coffey. When creating a process model (such as a flash tank) which involves an equilibrium state calculation connected to input or output process flow streams, take care in ordering the calls to these stream and thermodynamic parts. Specifically, calls to methods in the equilibrium calculation should be done *after* calls to methods in the streams. For example in MODEL dyn_flash.a4l:detailed_tray:

```
METHOD default_all;
   Qin := 0 {watt};
   RUN vapin.default_self;
   RUN liqin.default_self;
```

```
   RUN vapout.default_self;
   RUN liqout.default_self;
   RUN state.default_self;
   RUN default_self;
END default_all;
```

Here we see *state.default self* is called last. The part state shares information with vapout and liqout, naturally.

10.9 SUMMARY

adding our *standard* methods to a model definition

We have defined a set of standard methods for ASCEND models which we insist a modeler provide before we will allow a model to be placed in any of our model libraries. These are listed in Table 10-1. As should

Table 10-1 List of standard methods we insist be added for each of the types in our ASCEND library of type definitions

method	**description**
default_self	a method called automatically when any simulation is compiled to provide default values and adjust bounds for any locally created variables which may have unsuitable defaults in their ATOM definitions. Usually the variables selected are those for which the model becomes ill-behaved if given poor initial guesses or bounds (e.g., zero). This method should include statements to run the *default_self* method for each of its locally created (IS_A'd) parts. This method should be written first.
ClearAll	a method to set all the *.fixed* flags for variables in the type to *FALSE*. This puts these flags into a known standard state -- i.e., all are *FALSE*. All models inherit this method from the base model and the need to rewrite it is very, very rare.
specify	a method which assumes all the fixed flags are currently *FALSE* and which then sets a suitable set of *fixed* flags to *TRUE* to make an instance of this type of model well-posed. A well-posed model is one that is square (*n* equations in *n* unknowns) and solvable.
reset	a method which first runs the ClearAll method and then the *specify* method. We include this method because it is very convenient. We only have to run one method to make any simulation well-posed, no matter how its fixed flags are currently set. All models inherit this method from the base model, as with *ClearAll*.

Table 10-1 List of standard methods we insist be added for each of the types in our ASCEND library of type definitions

method	description
values	a method to establish typical values for the variables we have fixed in an application or test model. We may also supply values for some of the variables we will be computing to aid in solving a model instance of this type. These values are ones that we have tested for simulation of this type and found good.
bound_self	a method to update the *.upper_bound* and *.lower_bound* value for each of the variables. ASCEND solvers use these bound values to help solve the model equations. This method should bound locally created variables and then call bound_self for every locally created (IS_A'd) part.
scale_self	a method to update the *.nominal* value for each of the variables. ASCEND solvers will use these nominal values to rescale the variable to have a value of about one in magnitude to help solve the model equations. This method should rescale locally created variables and then call scale_self for every locally created (IS_A'd) part.

be evident from above, not *all* models must have associated methods; our first vessel model did not. It is simply our policy that models in our libraries must have these methods to promote model reuse and to serve as examples of best practices in mathematical modeling.

10.10 METHOD WRITING AUTOMATION

Just hit the button Library.Edit.Suggest methods and tweak the results.

ASCEND will help you write the standard methods. Writing most of the standard methods can be nearly automated once the declarative portion of the model definition is written. Usually, however, some minor tweaking of the automatically generated code is needed. In the Library window, the Edit menu has a "Suggest methods" button. Select a model you have written and read into the library, then hit this button.

In the Display window will appear a good starting point for the standard methods that you have not yet defined. This starting point follows the guidelines in this chapter. It saves you a lot of typing but it is a starting point only. Select and copy the text into the model you are editing, then tailor it to your needs and finish the missing bits. The comments in the generated code can be deleted before or after you copy the text to your model file.

If you have suggestions for general improvements to the generated method code, please mail them to us and include a sample of what the

generated code ought to look like *before* the user performs any hand-editing. We aim to create easily understood and easily fixed method suggestions, not perfect suggestions, because procedural code style tastes vary so widely.