CHAPTER 7 HOW TO DEFINE VARIABLES AND SCALING VALUES IN AN ASCEND MODEL

the purpose of this chapter

By now you have probably read Section 2, ”A Detailed ASCEND Example for Beginners: the modeling of a vessel,” on page 5 and seen an example of how to create a model using existing variable types in ASCEND. You found that variables of types area, length, mass, mass_density, and volume were needed and that they could be found in the library atoms.a4l. You want to know how to generalize on that; how to use variables, constants, and scaling values in your own models so that the models will be easier to solve.

This chapter is meant to explain the following things:

- The “Big Picture” of how variables, constants, and scaling values relate to the rest of the ASCEND IV language and to equations in particular. We’ll keep it simple here. More precise explanations for the language purist can be found in “The ASCEND IV language syntax and semantics” (syntax.pdf). You do not need to read about the “Big Picture” in order to read and use the other parts of this chapter, but you may find it helpful if you are having trouble writing an equation so that ASCEND will accept it.
- How to find the type of variable (or constant) you want. We keep a mess of interesting ATOM and CONSTANT definitions in atoms.a4l. We provide tools to search in already loaded libraries to locate the type you need.
- How to define a new type of variable when we do not have a predefined ATOM or CONSTANT that suits your needs. It is very easy to define your own variable types by copying code into an atoms library of your own from atoms.a4l and then editing the copied definition.
- How to define a scaling variable to make your equations much easier to solve.

7.1 THE BIG PICTURE: A TAXONOMY

As you read in Section 3, ”Preparing a model for reuse,” on page 25, simulations are built from MODEL and ATOM definitions, and MODEL and ATOM definitions are defined by creating types in an ASCEND language text file that you load into the ASCEND system. Figure 7-1 shows the types of objects that can be defined. You can see

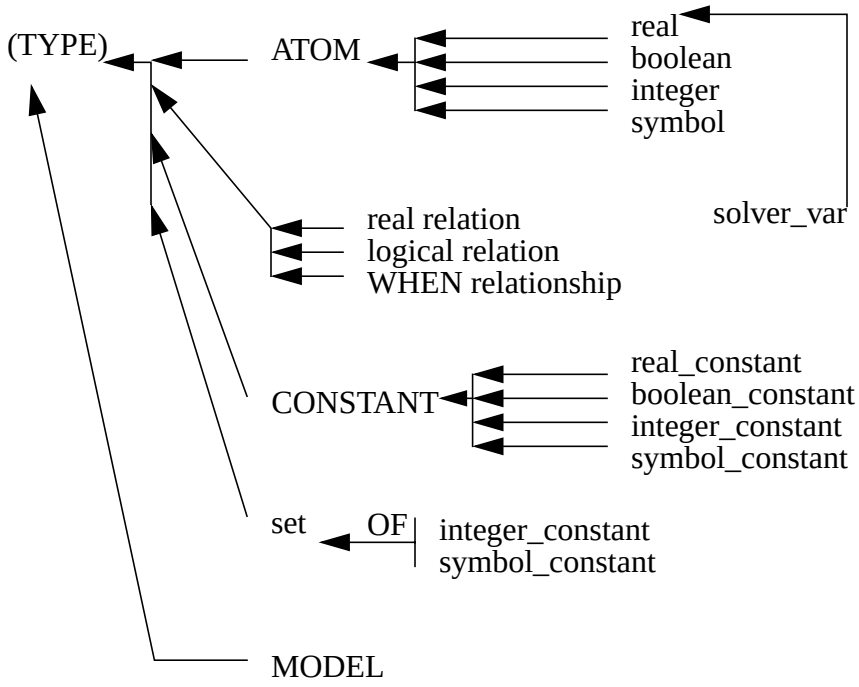

Figure 7-1 The Big Picture: How to think about variables

there are many more types than simply real variables used for writing equations. Some of these types can also be used in equations. You also see that there are three kinds of equations, not simply real relations. Throughout our documentation we call real relations simply “relations” because that is the kind of equation most people are interested in most of the time. Notice that “scaling values” do not appear in this diagram. We will cover scaling values at the end of this The major features of this diagram are:

ATOM

- Any variable quantity for use in relations, logical relations, or when statements or other computations. These come in the usual programming language flavors real, boolean, symbol, integer. Not all kinds of atoms can be used in all kinds of equations, as we shall explain when describing relations in a little bit. Atoms may be assigned values many times interactively, with the Script ASSIGN statement, with the METHOD := assignment operator,

or by an ASCEND client such as a solver.

An ATOM may have attributes other than its value, such as .fixed in solver_var, but these attributes are not atoms. They are subatomic particles and cannot be used in equations. These attributes are interpretable by ASCEND clients, and assignable by the user in the same ways that the user assigns atom values.

Each subatomic particle instance belongs to exactly one atom instance (one variable in your compiled simulation). This contrasts with an atom instance which can be shared among several models by passing the atom instance from one model into another or by creating aliases for it.

CONSTANT

- Constants are "variables" that can be assigned no more than once. By convention, all constant types in atoms.a4l have names that end in _constant so that they are not easily confused with atoms. A constant gets a values from the DEFAULT portion of its type definition, by an interactive assignment, or by an assignment in the a model which uses the :== assignment operator. Constants cannot be assigned in a METHOD, nor can they be assigned with the := operator.

Integer and symbol constants can appear as members of sets or as subscripts of arrays. Integer, boolean, and symbol constants can be used to control SELECT statements which determine your simulation's structure at compile-time or to control SWITCH and WHEN behavior during problem solving .

set

- Sets are unordered lists of either integer or symbol constants. A set is assigned its value exactly once. The user interface always presents a set in sorted order, but this is for convenience only. Sets are useful for defining an array range or for writing indexed relations. More about sets and their use can be found in syntax.pdf.

relationships

- Relations and logical relations allow you to state equalities and inequalities among the variables and constants in you models. WHEN statements allow you to state relationships among the models and equations which depend on the values of variables in those models. Sets and symbols are not allowed in real or logical relations except when used as array subscripts.

Real relations relate the values of real atoms, real constants, and integer constants. Real relations cannot contain boolean constants and atoms, nor can they contain integer atoms.

Logical relations relate the values of boolean atoms and boolean constants. The SATISFIED operator makes it possible to include real relations in a logical relation. Neither integer atoms and constants nor real atoms and constants are allowed in logical relations. If you find yourself trying to write an equation with integer atoms, you are really creating a conditional model for which you should use the WHEN statement instead. See Section 13, ”Creating Conditional Models in Ascend,” on page 131 to learn about how ASCEND represents this kind of mathematical model. There are also a real variable types, *solver_integer* and *solver_binary*, which are used to formulate equations when the solver is expected to initially treat the variable as a real value but drive it to an integer or 0-1 value at the solution. The integer programming features of ASCEND are described in a technical report by Craig Schmidt not yet available electronically. See system.a4l for elementary details.

Like atoms, real and logical relations may have attributes, subatomic particles for use by ASCEND clients and users. The name of a relation can be used in writing logical relations and WHEN statements.

WHEN statements are outside the scope of this chapter; please see Section 13, ”Creating Conditional Models in Ascend,” on page 131 or syntax.pdf for the details.

MODEL

- A model is simply a container for a collection of atoms, constants, sets, relations, logical relations, when statements, and arrays of any of these. The container also specifies some of the methods that can be used to manipulate its contents. Compiling a model creates an instance of it-- a simulation.

SOLVER_VAR

- The real atom type *solver_var* is the type from which all real variables that you want the system to solve for must spring. If you define a real variable using a type which is not a refinement of solver_var, all solvers will treat that variable as an a scaling value or other given constant rather than as a variable.

Solver_vars have a number of subatomic attributes (upper_bound, lower_bound, and so forth) that help solvers find the solution of your model. ATOM definitions specify appropriate default values for these attributes that depend on the expected applications of the atom. These attribute values can (and should) be modified by methods in the final application model where the most accurate problem information is available.

Scaling value

- A real which is not a member of the *solver_var* family is ignored by the solver. Numerical solvers for problems with many

equations in many variables work better if the error computed for each equation (before the system is solved) is of approximately size 1.0. This is most critical when you are starting to solve a new problem at values far, far away from the solution. When the error of one equation is much larger than the errors in the others, that error will skew the behavior of most numerical solvers and will cause poor performance.

This is one of the many reasons that scientists and engineers work with dimensionless models: the process of scaling the equations into dimensionless form has the effect of making the error of each equation roughly the same size even far away from the solution. It is sometimes easiest to obtain a dimensionless equation by writing the equation in its dimensional form using natural variables and then dividing both sides by an appropriate scaling value. We will see how to define an atom for scaling purposes in the last part of this chapter.

7.2 HOW TO FIND THE RIGHT VARIABLE TYPE

The type of real atom you want to use depends first on the dimensionality (length, mass/time, etc.) needed and then on the application in which the atom is going to be used. For example, if you are modeling a moving car and you want an atom type to describe the car's speed, then you need to find an atom with dimensionality length/time or in ASCEND terms L/T. There may be two or three types with this dimensionality, possibly including real_constants, a real scaling value, and an atom derived from solver_var.

Load atoms.a4l

The first step to finding the variable type needed is to make sure that atoms.a4l is loaded in your Library window from ascend4/models/atoms.a4l.

Find an ATOM or CONSTANT by units

The next step is to open the "ATOM by units" dialog found in the Library window's Find menu. This dialog asks for the units of the real variable type you want. For our example, speed, you would enter "feet/second," "furlongs/fortnight," "meter^3/second/ft^2" or any other combination of units that corresponds to the dimensionality L/T.

If the system is able to deduce the dimensionality of the units you have entered, it will return a list of all the currently loaded ATOM and CONSTANT definitions with matching dimensions. It may fail to understand the units, in which case you should try the corresponding SI units. If it understands the units but there are no matching atoms or constants, you will be duly informed. If there is no atom that meets your needs, you should create one as outlined in Section 7.3.

Selecting the right type

The resulting list of types includes a Code button which will display the definition of any of the types listed once you select (highlight) that type with the mouse. Usually you will need to examine several of the alternatives to see which one is most appropriate to the physics and mathematics of your problem. Compare the default, bounds, and nominal values defined to those you need. Check whether the type you are looking at is a CONSTANT or an ATOM.

You now know the name of the variable type you need, or you know that you must create a new one to suit your needs.

7.3 HOW TO DEFINE A NEW TYPE OF VARIABLE

In this section we will give examples of defining the atom and constant types as well as outline a few exceptional situations when you should NOT define a new type. More examples can be found and copied from atoms.a4l. You should define your new atoms in your personal atoms library.

Saving customized variable types

The normal location for this personal library is in the user data file ~\ascdata\myatoms.a4l. This file contains the following three lines and then the ATOM and CONSTANT definitions you create.

```
REQUIRE “atoms.a4l”; (* loads our atoms first *)
PROVIDE “myatoms.a4l”; (* registers your library *)
(* Custom atoms created by <insert your name here> *)
```

If you develop an interesting set of atoms for some problem domain outside chemical engineering thermodynamics, please consider mailing it to us through our web page.

The user data directory ~/ascdata may have a different name if you are running under Windows and do not have the environment variable HOME defined. It may be something like C:\ascdata or \WINNT\Profiles\Your Name\ascdata. When ASCEND is started, it prints out the name of this directory.

When you write a MODEL which depends on the definition of your new atoms, do not forget to add the statement

```
REQUIRE “myatoms.a4l”;
```

at the very top of your model file so that your atoms will be loaded before your model definitions try to use them.

7.3.1 A NEW REAL VARIABLE FOR SOLVER USE

Suppose you need an atom with units {dollar/ft^2/year} for some equation relating amortized construction costs to building size. Maybe this example is a bit far fetched, but it is a safe bet that our library is not going to have an atom or a constant for these units. Here is the standard incantation for defining a new variable type based on solver_var. ASCEND allows a few permutations on this incantation, but they are of no practical value. The parts of this incantation that are in *italics* should be changed to match your needs. You can skip the comments, but you **must** include the units of the default on the bounds and nominal.

```
ATOM amortized_area_cost
REFINES solver_var DEFAULT 3.0 {dollar/ft^2/year};
   lower_bound := 0 {dollar/ft^2/year};
   (* minimum value *)
   upper_bound := 10000 {dollar/ft^2/year};
   (* maximum value for any sane application *)
   nominal := 10 {dollar/ft^2/year};
   (* expected size for all reasonable applications*)
END amortized_area_cost;
```

In picking the name of your atom, remember that names should be as self-explanatory as possible. Also avoid choosing a name that ends in *_constant* (as this is conventionally applied only to CONSTANT definitions) or *_parameter.* Parameter is an extremely ambiguous and therefore useless word. Also remember that the role a variable plays in solving a set of equations depends on how the solver being applied interprets *.fixed* and other attributes of the variable.

Exceptions

If an atom type matches all but one of the attributes you need for your problem, say for example the upper_bound is way too high, use the existing variable type and reassign the bound to a more sensible value in the *default_self* method of the model where the variable is created. Having a dozen atoms defined for the same units gets confusing in short order to anyone you might share your models with.

The exception to the exception (yes, there always seems to be one of those) is the case of a lower_bound set at zero. Usually a lower_bound of zero indicates that there is something inherently positive about variables of that type. Variables with a bound of this type should not have these physical bounds expanded in an application. Another example of this type of bound is the upper_bound 1.0 on the type *fraction*.

For example, negative temperature just is not sensible for most physical systems. ASCEND defines a *temperature* atom for use in equations involving the absolute temperature. On the other hand, a temperature difference, delta T, is frequently negative so a separate atom is defined. Anyone receiving a model written using the two types of atoms which both have units of {Kelvin} can easily tell which variables might legitimately take on negative values by noting whether the variable is defined as a *temperature* or a *delta_temperature*.

7.3.2 A NEW REAL CONSTANT TYPE

Real constants which do not have a default value are usually needed only in libraries of reusable models, such as components.a4l, where the values depend on the end-user's selection from alternatives in a database. The standard incantation to define a new real constant type is:

```
CONSTANT critical_pressure_constant
REFINES real_constant DIMENSION M/L/T^2;
```

Here again, the *italic* parts of this incantation should be redefined for your purpose.

Universal exceptions and unit conversions

It is wasteful to define a CONSTANT type and a compiled object to represent a *universal* constant. For example, the thermodynamic gas constant, R = 8.314... {J/mole/K}, is frequently needed in modeling chemical systems. The SI value of R does not vary with its application. Neither does the value of π. Numeric constants of this sort are better represented as a numeric coefficient and an appropriately defined unit conversion. Consider the ideal gas law, PV = NRT and the ASCEND unit conversion {GAS_C} which appears in the library ascend4/models/measures.a4l. This equation should be written:

```
P * V = n * 1.0{GAS_C} * T;
```

Similarly, area = pi*r^2 should be written

```
area = 1{PI} * r^2;
```

The coefficient 1 of {GAS_C} and {PI} in these equations takes of the dimensionality of and is multiplied by the conversion factor implied by the UNITS definition for the units. If we check measures.a4l, we find the definition of PI is simply {3.14159...} and the definition of GAS_C is {8.314... J/mole/K} as we ought to expect.

For historical reasons there are a few universal constant definitions in atoms.a4l. New modelers should not use them; they are only provided to support outdated models that no one has yet taken the time to update.

7.3.3 NEW TYPES FOR INTEGERS, SYMBOLS, AND BOOLEANS

The syntax for ATOM and CONSTANT definitions of the non-real types is the same as for real number types, except that units are not involved. Take your best guess based on the examples above, and you will get it right. If even that is too hard, more details are given in syntax.pdf.

7.4 HOW TO DEFINE A SCALING VARIABLE

A scaling variable ATOM is defined with a name that ends in *_scale* as follows. Note that this ATOM does not refine solver_var, so solvers will not try to change variables of this type during the solution process.

```
ATOM distance_scale REFINES real DEFAULT 1.0{meter};
END distance_scale;
```

ASCEND cannot do it all for you

ASCEND uses a combination of symbolic and numerical techniques to create and solve mathematical problems. Once you get the problem close to the solution, ASCEND can internally compute its own scaling values for relations, known elsewhere as “relation nominals”, assuming you have set good values for the *.nominal* attribute of all the variables. It does this by computing the largest additive term in each equation. The absolute value of this term is a very good scaling value.

This internal scaling works quite well, but not when the problem is very far away from the solution so that the largest additive terms computed are not at all representative of the physical situation being modeled. The *scale_self* method, which should be written for every model as described in Section 10.4.4, should set the equation scaling values you have defined in a MODEL based on the best available information. In a chemical engineering flowsheeting problem, for example, information about a key process material flow should be propagated throughout the process flowsheet to scale all the other flows, material balance equations, and energy balance equations.

Scaling atom default value

The default value for any scaling atom should always be 1.0 in appropriate SI units, so that the scaling will have no effect until you assign a problem specific value. Multiplying or dividing both sides of an equation by 1.0 obviously will not change the mathematical

behavior, but you do not want to change the behavior arbitrarily either--you want to change it based on problem information that is not contained in your myatoms.a4l file.