Object-Oriented Software Architecture for Large-Scope Numerical Simulation

David Gedeon
Gedeon Associates
Athens, Ohio 45701

September 1996

An article on Sage submitted to Computational Science and Engineering magazine, but rejected for publication. Sigh...

Fresh out of graduate school in 1975 I found myself working in a small engineering firm developing stirling-cycle machines. For those of you who forget, the stirling cycle is a closed thermodynamic cycle involving a high-pressure gas such as helium alternately compressed and expanded in two piston cylinders connected by heat exchangers through which the gas flows back and forth. The heat exchangers change the temperature of the gas as it flows from one piston cylinder, the so-called compression space, to the other, the expansion space. The point of all this is to convert thermal energy to mechanical work or mechanical work to thermal energy in the form of refrigeration. The details are not important here. What is important is that my job was to develop a thermodynamic modeling program for such devices using the tool of the time, which was FORTRAN IV running in batch mode on the local university mainframe computer. I had been trained in mathematics and physics, so this seemed like a reasonable idea.

Reasonable or not, I eventually did develop such a program. It was based on a one-dimensional finite-difference model and while it certainly obeyed some of the more obvious laws of physics — for example, thermodynamic efficiency was generally less than the theoretical Carnot limit — the model suffered from several omissions of detail with the result that predicted power levels and efficiencies were not always realized in the laboratory. In fact,

on several occasions, machines perfectly reasonable from a modeling point of view failed to run at all because of something like mechanical friction missing from the model.

Thus began my life's work, which has continued more-or-less along these lines these last 20 years. While I am still developing software tools for modeling stirling-cycle machines, these days the machines are more apt to be cryogenic refrigerators for electronics applications rather than, say, automotive engines, which were the fashion of the time. And the mainframe computer has given way to a desktop computer. And my language of choice is now Borland International's Delphi, which is a visual programming environment based on their Object Pascal language. Also, I now work for myself as an independent software developer and consultant rather than as an employee of any one company.

So my work has brought me into contact with a large assortment of practicing engineers mostly in the USA and Europe. And what do I find about these engineers? First of all, they are always changing things. My original models were for fixed geometries, but this was never enough. Someone always wanted to stick in a valve here, a manifold there or introduce some new gizmo that I hadn't thought to include. Secondly, they are driven by impure motives such as cost reduction, pressure-vessel regulations, customer requirements and the whims of evil project managers. Mathematically speaking, they have constraints and they need to optimize their machines within these constraints. Thirdly, the stirling-cycle part was generally only a small piece of a total system. So it simply would not do, for example, to design the perfect stirling-cycle machine operating at 1 Hz with a 10 cm piston stroke if it had to be driven by a 60 Hz linear motor with 10 mm stroke. What was I to do?

Fortunately, about that time the art of computer programming had evolved object orientation and the visual interface. The concept of software *objects* was intriguing because it suggested a way out of the trap of micromanaging all aspects of a numerical model at the time of programming. It would now be possible to develop a framework in which the user could interconnect relatively simple component parts into a complete system. And the visual interface could complete the illusion by giving visual presence to these software objects.

Possible, but not easy. There I was, in the early days, armed with a few snippets of object-oriented code, convinced that herein lay the path to modeling a complete Boeing 747 aircraft, but yet wondering where all the details

would go. Certainly there must be a few details in a Boeing 747? As I progressed, my ideas about object oriented modeling became more complicated while the machines to be modeled became simpler. And so it went for about three years, until now the machines I can model are about as complicated as a bicycle.

Sage from a User's Perspective

The name of the program that models machines about as complicated as a bicycle — namely stirling machines — is Sage, which does not stand for anything in the usual sense of a computer program, but rather suggests its behavior as somewhat of a wise overseer of model components. Here are the main things Sage does:

- Allows a user to create a custom engineering model by graphically selecting component parts from a palette, dropping them into a window and connected them together as required.
- Manages the interconnected system of components so the user can easily change data inputs, solve the underlying equations of the system and view the results.
- Supports interactive system-wide optimization of an arbitrary set of input variables subject to an arbitrary number of constraints (nonlinear equality or inequality) with an arbitrary objective function to be minimized or maximized.

Notice that nothing said about Sage so far is specific to any particular class of model components. Sage would be as happy to work with electrical or chemical models as mechanical models. In fact, with any system comprising a number of well-defined components that can be interconnected together. It is just that because of my limited resources and expertise, the only model components available at the moment apply to stirling-cycle machines.

Sage runs under the Microsoft Windows environment and employs both graphical and text-based interfaces depending on what the user is focused on at the time. By way of example, figure (1) shows how a stirling cryogenic refrigerator appears in Sage from the graphical point of view. A user might use this point of view when creating new models or tracing interconnections

between model components. Figure (2) shows a text-based view of a particular model component, the important viewpoint for specifying input variables or monitoring output values.

Sage from a Programmer's Perspective

So how does Sage work? At this point a broad sketch may be useful.

First, there are the model components themselves which form a hierarchy of software object classes. There is an ultimate ancestor class from which all model components descend — basically just a container for whatever variables the model will contain, with a number of methods (functions and procedures) and data structures (dynamically-sized lists or collections) for managing these variables. Subsequent descendants add the functionality for whatever it is the model component is supposed to be modeling.

The model variables themselves are considerably more elaborate than the variables employed in a standard programming language such as FORTRAN, Pascal or C. In addition to storing a numerical value corresponding to some physical quantity, these variables also manage themselves as much as possible, rather more like actors on a stage than puppets managed from on high. Variables, too, form a hierarchy of object classes.

Next there are software layers that know how to process user commands to create model components at run time and organize these components into a whole. At the first level of organization, model components reside within a dynamically-allocated tree data structure. There is always a root-level model component. Then, depending on what is being modeled, there are some number of child components, each with its own child components, and so forth. Model trees are convenient from a programmer standpoint because they can be scanned in an orderly sequence for purpose of creating data files, output listings, collections of things, performing searches or carrying out various operations. Figure 3 shows this structure for the cryogenic refrigerator depicted in figure 1. In another level of organization, model components are interconnected across their boundaries for purposes of exchanging forces, fluid flows, etc. Boundary connections bring a model to life, much as the connections among cells bring a biological organism to life. They are also helpful to the user in visualizing the model's purpose.

And then, there are yet more software layers, generally object classes, that manage tasks such as equation solving and optimization. The impor-

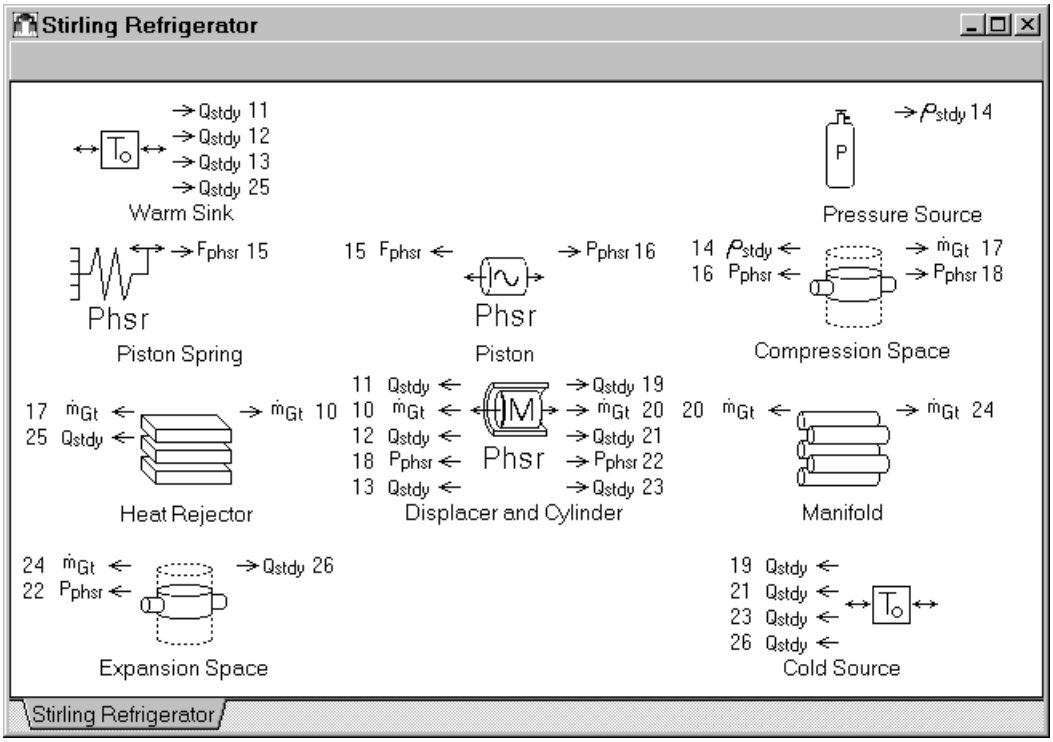

Figure 1: A stirling-cycle refrigerator assembled from individual model components. Graphical icons represent the components while labeled and numbered arrows represent various types of boundary connections: $\dot{m}_{G_t}$ = gas flow. Q_{stdy} = steady heat flow. P_{phsr} = phasor (sinusoidal) pressure between volume displacements. ρ_{stdy} = mean density establishing pressure reference.

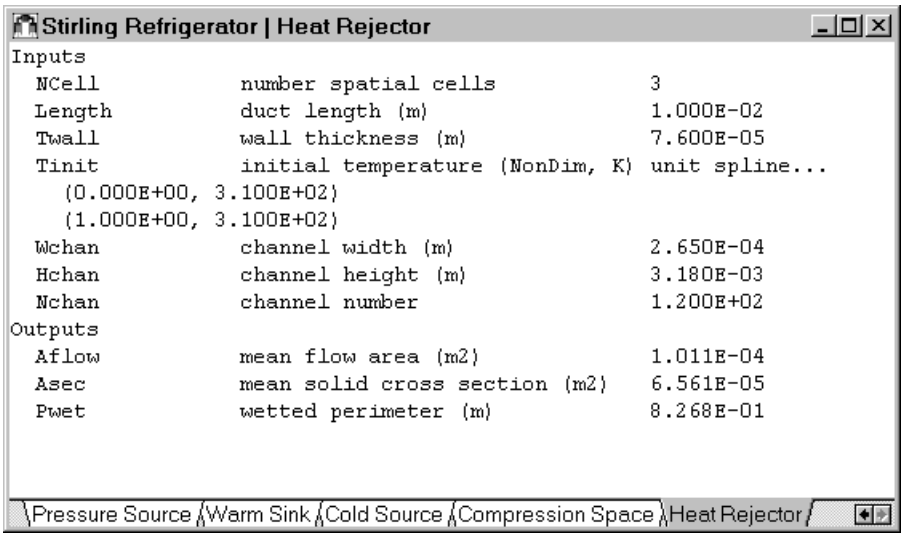

Figure 2: Text window for a single model component within a stirling-cycle refrigerator.

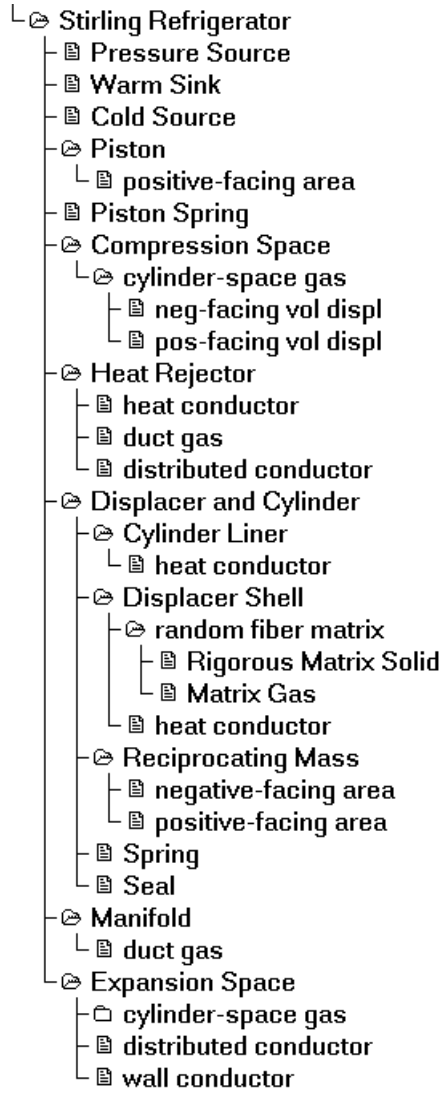

Figure 3: The tree data structure for the model components comprising a stirling-cycle refrigerator.

tant thing about these entities is that they must access model components and variables in a completely generic way without knowing or caring about specific implementation details.

Evidently, much of the work in designing Sage was in deciding the best way to delegate responsibility for the various tasks of model management among the various software object classes. This was a little like building a transcontinental railroad from both ends at once, with logical rails not always meeting up in the middle. Part of the trouble early on was a too-rigid insistence on the object-oriented paradigm. For example, it did not dawn on me for some time that there was no way a model component could be completely responsible for solving, much less optimizing, itself within a total system. Somewhere there had to be a level of software responsible for overall management.

Smart Variables

Beginning now a more detailed look at things, we might as well start with the fundamental building blocks of model components — the variables.

Behavior Types

The ultimate ancestor of all variables is a data structure known as TVar that does not yet know what sort of numerical data it stores, but only whether or not that data is valid and what to do about if it is not, depending on the way the variable may be used in a model. There are four possible ways: as a *constant*, *independent* variable, *dependent* variable or *implicit* variable. This information, together with the state of validity are encoded within a *flag* byte. Collectively, constants and independent variables form the inputs of a model component. Dependent and implicit variables form the outputs.

Constant

A *constant* holds a numerical value (real, integer, complex, ...) that remains fixed during solution or optimization but may be changed by the user from time to time. Constants generally hold normalization quantities or computational grid dimensions, both of which are at a conceptually higher level than the model itself. A constant is always valid.

Independent

An *independent* variable is like a constant except that its value may also be set by an external driver, such as the optimizer. Independent variables generally hold values for physical dimensions or model parameters. An independent variable is also always valid.

Dependent

A *dependent* variable holds a numerical value that is dependant on other variables, possibly other dependent variables, but eventually on non-dependent variables. All dependent variables have a mechanism — an evaluation function — by which they know how to evaluate themselves explicitly. After evaluation, a dependent variable becomes valid but it may become invalid subsequently if the variables upon which it depends change.

Implicit

Finally, an *implicit* variable holds a numerical value that is to be iteratively refined by an external solver to satisfy some implicit relationship among model variables, usually including itself. All implicit variables have an explicit evaluation function for providing their initial value and an implicit evaluation function for evaluating the degree to which their implicit relationship is satisfied. Implicit relationships are generally much easier to formulate and extend to ever more complex systems than explicit relationships. Technically speaking, an implicit variable is always valid after it is first initialized, even though it may not actually satisfy all the implicit relationships which reference it until after the solver converges.

Evaluation Logic

As with variables in any programming language it is possible to assign or reference the numerical value of a Sage variable. But there are also side effects.

The side effects of assigning a variable's numerical value depend on its behavior type. If constant, independent or implicit, then the variable calls upon its parent model component to invalidate all the dependent variables that could possibly depend on the new value. If dependent, then the variable calls upon its parent model component to push a pointer to itself onto a

temporary stack of valid variables so they can be easily invalidated later if required. In all cases, the variable sets its own *valid* flag true.

The side effects of referencing a variable's numerical value depend on its current state of validity. If valid, then there are no side effects — the stored value is just returned. If invalid, then prior to returning the stored value the variable first validates itself by assigning its numerical value to the result returned by its internal evaluation function, according to the logic of the assign process just discussed. Dependent variables may have to validate themselves a lot as model inputs change or during the iterative solution process. Implicit variables generally have to validate themselves only once at the time of first reference. Constants and independent variables are born valid and always remain that way.

This evaluation logic may seem a needlessly round-about way of doing things. But it does save time. In some cases a lot of time. Consider, for example, a dependent variable whose value depends on a lengthy calculation involving many lower-level variables whose values mostly remain fixed. Then, rather than repeat this evaluation process each time the first variable is referenced, it is much faster if it can just cut to the chase and return its stored value so long as it is guaranteed valid. The above logic supports this behavior.

One logical thread left unexplored is how a model component knows which dependent variables, possibly in other model components, depend on a given constant, independent or implicit variable within itself? This could get messy if model components were to keep track of this on the individual-variable level. What they do instead is keep track on the model-component level. The basic container class for variables, from which model components descend, is known as TVarSet. Each TVarSet instance contains a collection known as the *Affected VarSet Collection*, or AVC for short. The AVC contains pointers to all TVarSet instances with dependent variables affected by some constant, independent or implicit variable of itself. The AVC is created at runtime by accessing the dependent variables of the entire model tree one by one, allowing each evaluation-function call to propagate until it ultimately terminates in references to constants, independent or implicit variables, which announce this fact. Then, a pointer to the TVarSet instance owning the original dependent variable is stored in the AVC of each TVarSet instances owning a terminally referenced variable. Of course, each time the model structure changes (as when new components are added) the AVC collections must be rebuilt.

Names

All variables have a field which holds a pointer to their name. A name is useful for referencing the variable in user-defined algebraic expressions (discussed later) and for presenting the variable for input and output purposes. Often a variable is completely hidden from the user, in which case the name is a space-saving nil pointer. But when a name is required it actual consists of a simple data structure containing two strings: an identifier and a short definition. For example "Lnorm" might be the identifier for a variable with the definition "length scale". This identifier is separate from the identifier by which the programmer refers to the same variable (it is for run-time purposes while the other is for the Delphi compiler) but its purpose and syntax are similar.

Numerical Types

Only after all the above behaviors are defined do variable classes get down to dealing with particular numerical data types. Descended from TVar are a number of specialized variable classes for the different sorts of numerical values convenient in numerical models.

Reals

Real variables have an extended-precision floating-point data field. Variables of this type are the most common choice for all behavior types. When used as constants or independent variables, there are variations that restrict their values to non-negative, strictly positive or closed-interval ranges. These have a built-in ability to properly validate user input. When used as an implicit variable there is a variation designed to restrict the solution from changing too quickly during the solving process, thereby preventing convergence instability.

There is even a special sort of real-valued dependent variable whose value comes from a string expression supplied by the user interactively at run time. This string expression is stored as the data field of the variable and might look something like: Pressure * Diameter / (2 * WallThickness), which resembles an expression used in any typical programming language, except the variable identifiers referenced are those of other variables in the model at run time. A user might decide that such an expression is appropriate for a variable

named Stress, an identifier also user supplied. Such variables are more than just a luxury. They are essential for enabling a user to extend the model in ways not envisioned by the programmer, especially during optimization. Optimization will come up as a separate topic later on.

Integers

Integer variables have an integer data field. Variables of this type are used for constants or dependent variables but not independent or implicit variables because the solver or optimizer cannot deal with discrete integer values. There are variations that restrict their values within a closed-interval range. And there is a special variation used only for constants that establish the dimensions of repetitive data structures such as computational grids. Changing the value of such a variable sets into motion a mechanism that dynamically re-builds any affected data structures.

Complex or Phasor

Complex variables have two extended-precision fields useful for representing complex numbers or phasors in application models. They may behave as constants, independent or dependent variables, but not implicit variables because of assumptions made by the solver. Phasor variables are minor variations of complex variables that perform I/O in polar rather than Cartesian format. Complex variables are especially useful for models containing linear equation systems and sinusoidally varying quantities.

Splines

Spline variables have a pointer to a data structure known as TSpline as their data field. Splines are used for representing cubic splines as constants, independent or dependent variables. The TSpline data structure consists of an arbitrary number of extended-precision interpolation pairs of the form $(T, F(T))$ where T represents an independent variable such as temperature and $F(T)$ a dependent variable such as some temperature-dependent property. Built into TSpline is an interpolation function to return $F(T)$ for intermediate values of T. One variation of the basic spline variable restricts the independent variable to the range $0 \leq T \leq 1$. Another variation transforms both T and $F(T)$ by the log function before cubic-spline interpolation as if

interpolation were done on a log-log plot. And yet another variation transforms $F(T)$ only by the log function before cubic-spline interpolation as if interpolation were done on a semi-log plot. Spline variables are especially handy for specifying things like thermophysical properties as a function of temperature or — if the independent variable represents position along the axis of a computational domain — any sort of initial value or boundary condition as a function of position.

Fourier Series

Fourier series variables have a pointer to a data structure known as TFSeries as their data field. Fourier series are used for representing Fourier series as constants, independent or dependent variables. The TFSeries data structure consists of an extended-precision mean value, with a number of extended-precision cosine and sine coefficients representing a real-valued periodic function $F(\omega t)$ with period 2π. Built into TFSeries is an evaluation function to return $F(\omega t)$ for any value t. There is a minor variation which does I/O of harmonic coefficients in polar rather than Cartesian format — i.e. replacing (a_n, b_n) in $a_n \cos n\omega t + b_n \sin n\omega t$ with (c_n, r_n) in $c_n \cos(n\omega t + r_n)$. Fourier series variables are very convenient for representing outputs of time-periodic models.

Enumerated

There are also variable classes that store more complicated data types used as constants. The user selects possible instances of such variables from an enumerated list rather than specifying a possibly-large set of individual values for the underlying data type. A good example would be the gas-property variable whose underlying data type TGas contains several real or spline-valued fields corresponding to the thermophysical properties of a particular gas. The user would specify such a variable by selecting a name such as helium from a list. Entering detailed thermophysical properties for a suitable list of such variables is a data-base management task outside the scope of Sage.

Referencing and Assigning Non-Reals

Real variables may be directly referenced by name in user-defined expressions or, if independent, assigned by an external driver such as the optimizer. It is very convenient if the real parts of other numerical types can be likewise referenced or assigned. For example, if an independent complex-valued variable X represents piston displacement, then a user might want to reference or assign the amplitude of X. The Sage complex-variable class allows this through use of the Amp sub-identifier attached to the main variable identifier, as in X.Amp. Sub-identifiers are generally available for all the real-valued subfields a user might be interested in accessing within the the complex, spline or Fourier series numerical types.

Input and Display

Every type of variable is responsible for managing the input of its numerical data in the event it is used as a constant or independent variable. The same goes for its output format too, regardless of the mode of use. Although, only variables with non-nil names are ever called upon to display their output.

Input is in the form of an interactive dialog, the details of which are necessarily operating-system specific. But, generally, the presentation is in the form of a dialog which displays the identifier and definition strings and allows the user to modify the numerical data value, be it real, integer, or whatever.

Output is in the form of a data structure containing the identifier and definition strings followed by a value string with an optional dynamically-sized collection of sub-value strings for the more complicated data types.

Evaluation Functions

In support of the evaluation logic previously discussed, there are three evaluation functions that variables may contain: Eval, Fimpl and Norm. The use of these evaluation functions in the context of the various evaluation behaviors can be best summarized in the form of a table:

	constant	independent	dependent	implicit
Eval	—	—	evaluation	initialization
Fimpl	—	—	—	solution
Norm	—	normalization	normalization	normalization

Depending on allowed behavior types, a variable class may not define all these evaluation functions. And it is not quite correct to say that the functions are *contained* in the variables. What is contained, actually, are just function addresses as data fields, possibly nil addresses if particular evaluation functions are not required in a given instance. These addresses are supplied as constructor arguments at the time of variable creation.

In the Delphi Language, the above evaluation functions are accessed through what are known as properties. Properties are referenced like ordinary data fields. For example, Val is the property associated with the Eval function. A reference to Length.Val for a variable named Length calls its Eval function or just returns its stored numerical value, according to the evaluation logic previously discussed. This is all transparent to the programmer who just references Length.Val as if it were a variable in a conventional programming language. Properties may also be assigned like ordinary data fields. For example, an assignment to Length.Val would set its value field and also carry out any of the side effects previously discussed. Part of the power of Delphi properties is that the Val property can be overridden for each variable class to have the same type as the numerical data field. So, for example, one might assign or reference a complete Fourier series directly through the Val property.

Explicit Evaluation

The Eval function returns the current value of a dependent variable or the initial value of an implicit variable. For example, figure 4 contains a the Delphi code for a dependent-variable evaluation function, one of several such functions in a model component dedicated to defining internal geometries for heat exchanger ducts. Although this example is particularly simple, evaluation functions are generally no more than a few lines of code long — even for variables within nodes of computation grids representing complex physical phenomena, such as compressible gas flow.

While each evaluation function may be relatively simple the effect of many taken together may be quite complex. A typical evaluation function will generally reference the Val property for other variables, some of them dependent variables. If these turn out to be invalid at the time of evaluation, then they too will call upon their evaluation functions, referencing yet more Val properties. This process may go on for a while, cascading throughout a vast network of variables, until eventually the first evaluation function

```
function TubDctAflowEval(AParent: TTubDct): Extended; far;
begin
  with AParent do
    Result:= Ntube.Val * 0.25 * Pi * Sqr( Dtube.Val );
end;
```

Figure 4: A simple evaluation function accessed by a dependent variable name Aflow (flow area) contained within a particular sort of heat-exchanger duct, the purpose being to calculate itself in terms of independent variables Ntube (tube number) and Dtube (tube diameter).

returns. But complex as this process may be, the programmer does not need to worry about it directly. It is automatically managed by the architecture.

Implicit Evaluation

The Fimpl function specifies the implicit relationship to be satisfied for an implicit variable. For example, Newtons second law "Force = Mass × Acceleration " might be viewed as an implicit relationship for purposes of solving Mass, especially if were not so simple to isolate Mass on one side of the equation. Say, for example, if Force somehow depended on Mass in some especially perverse model. The actual evaluation function would return the result "Force - Mass × Acceleration ", which is zero when Newton's law is satisfied. Presently, Fimpl is always a real-valued function with the returned value measuring the degree by which the implicit relationship is not satisfied. A Delphi code example of an implicit-relationship function is shown in figure 5. The task of simultaneously zeroing the Fimpl functions for all implicit variables of the entire model is the job of Sage's nonlinear equation solver.

Normalization

The Norm function returns a real-valued normalization constant used for scaling the value of an independent, dependent or implicit variable to the order of one so that it may be more conveniently dealt with by Sage's optimizer or solver. It is a function, rather than just a real data field, to allow the possibility of referencing one or more constants in an expression similar to one that might be used for the Eval function.

Data Streaming

Finally, every class of variable is responsible for loading and storing its internal data to or from a storage stream. A stream is a free-form linear concatenation of arbitrary data types generally read or written sequentially from beginning to end. Streams are general concepts supported within the Delphi programming environment and are usually associated with disk files, although streams in random-access memory are also possible.

One of the hard parts about streaming variables concerns reading and writing the addresses of evaluation functions and the like which are stored as data fields within the variables. These cannot be streamed directly because, of course, they are apt to change depending on where the program is loaded in memory. So they are streamed indirectly by name, or rather by unique integers associated with each such function. This necessitates a bit of housekeeping code where the address of each evaluation function is associated with its corresponding integer storage value. This code is executed within a virtual (polymorphic) method overridden for each TVarSet containing variables with evaluation functions.

Variable Collections

Variables acting alone are not worth much. They must be organized together into increasingly more complicated structures before they can hope to do any worthwhile modeling. The analogy of primitive cellular life forms swimming around in a primordial sea comes to mind. Only after these cells finally organize into complex organisms do we get the interesting results. This analogy with life, by the way, has continued to afflict me all through the development of Sage. I've come to rely on it as a guide to when I have done something right.

But at any rate, the first level of organization of variables into higher forms is TVarSet, which has already found its way into the previous discussion. Essentially, TVarSet serves as an indexed collection for the variables it owns, although being an abstract collection, it does not yet own any variables. Descendants of TVarSet will, of course, own variables. Sometimes these variables will be nameless entities referenced only by their index number. Sometimes they will be have individual identifiers so they may be most conveniently referenced during programming.

The purpose of TVarSet is to implement a number of low-level variable management routines that it will pass on to its descendants. In support of this it defines some useful data structures, the following of which will already make sense in terms of the previous discussion:

Name	Structure	Purpose
Parent	pointer	Parent TVarSet instance.
ChildVarSetC	collection	Child TVarSet instances.
AffecVarSetC	collection	TVarSet instances whose dependent variables are affected by constants, independent or implicit variables of self.
ValidDepStack	stack	Currently valid dependent variables.
VarRelator	linked list	(integer, address) pairs for loading and storing evaluation functions to a stream.

The more complex fields of TVarSet are actually pointers to the data structures in question, with the data-structures themselves (collection, stack, linked list) residing in dynamically allocated memory.

The ChildVarSet collection is worthy of note. It is the basis for the eventual tree structural organization of model components. Many of the procedural methods built into TVarSet are designed to work in a recursive matter down through all generations of child TVarSet instances, starting from the level at which the method is first accessed.

Store Thyself

One of the interesting things about TVarSet instances is they have the ability to load and store themselves to or from a stream, usually associated with a disk file. This is not as hard as it might sound because all variables and most data structures within are already streamable objects that know how to load and store their own internal data. So, for example, the TVarSet store method proceeds something like this:

- asks all owned variables to store themselves
- asks other owned data structures to store themselves

- asks all children in ChildVarSetC to store themselves

Evidently, recursion comes into play in the last step. The companion load method proceeds along similar lines. The resulting stream is a total mess to the untrained eye but makes complete sense when read or written in the order in which the various load and store methods are called.

But when it comes time to re-load objects from a stream, how does Sage know ahead of time which object class methods to call upon to do the loading? The answer is encoded in the stream. Each streamable object has a unique class-identifier number associated with it by which it is recognized in the stream. This number immediately precedes the object's data. Loading from the stream involves reading this identification number, scanning a lookup table for the appropriate object class, then calling on the load method of that class to read its data. This is mostly automatic, except registering object classes and their stream identifiers is a necessary initialization step.

A somewhat different problem with a similar solution is how to store and load pointers to object instances owned elsewhere. Sometimes this is inevitable as when a descendant of TVarSet uses a locally-declared pointer to refer to a variable owned by another TVarSet instance. It would not do to load and store duplicate copies of the variable. Their data fields would quickly get out of sync. Instead, such pointers are stored indirectly, usually through the associated index in some collection, and re-generated on loading by referencing that collection. This sometimes requires use of a temporary *fixup* list when, for example, the collection in which the original object resides is not yet loaded from the stream. In a fixup list, all pointer variables that will eventually reference the yet-to-be loaded object are temporarily configured into a linked list, pointing to each other in succession, with the beginning node of the list stored in some temporary variable. Then after the load sequence progresses to the point where the object to be pointed at has a firm address, the fixup list is traversed and all references *fixed up*, after which the list vanishes.

Having a way to store TVarSet data to disk files is essential. And streams are a powerful and convenient way to do so. But they have their downside. The downside is that streams make object class maintenance more difficult because any modification that introduces a new data field, such as a new variable, suddenly makes all previously stored instances of its former self unloadable. At least not without some kind of data-conversion process. And because of the nature of streams, this data conversion process typically in-

volves compiler directives within each modified object's load method to deal with data fields on a case-by-case basis, rather than a separate self-contained code module.

Model Components

An immediate descendant of TVarSet is TModel, the ancestor of all model components. The TModel class support a number of methods intended for user interaction, as well as inheriting all TVarSet methods. Also fleshed out are the rudimentary structures and behaviors required for connecting to other TVarSet instances across their boundaries.

Being Born

Model components are designed to be created at run time, at the drag-and-drop whim of the user, rather than as pre-ordained by the programmer. So where do they come from? They come from seeds. And how do these seeds sprout? They germinate. Again, the analogy with life — plant life, at any rate.

The way this works is that model component instances are created in a two step process. The first step gives them only a visual presence so that they can appear as a bitmap in a model component palette. The second step is germination, after which they completely flesh out all their variables and other data structures to do whatever it is they do. But they still appear as a bitmap, only now in a special window for *living* model components. Once alive, a model component has the ability to fill the palette with seeds of its own, representing possible child model components. Here the life analogy breaks down because there is no mating process required to beget child model components. And the child model components are generally not identical to the parent, but rather more like attachments or subsystems. For example, a child model to a gas domain in Sage might be an inlet through which gas flows.

Sometimes the seeds created by a given model component contain pointers that reference variables in the parent component. This is useful when, say, a seed for a child-component containing a computational grid needs to know the number of nodes to create upon germination according to the value of a variable owned by the parent. Even after germination, this umbilical cord

remains uncut so that child model components are rarely completely independent of their parents. In fact, child model components frequently call upon their parents to perform certain functions, and the other way around.

Connections

After germination, a model component is a self contained entity with a valid internal solution in the absence of external influences. For example, a component representing a thermal conduction path just produces an isothermal state as its solution if unconnected to a source and sink temperature. But the interesting thing is what happens when one connects two model components together. For example, a temperature source to a conduction path. The connection process dynamically creates a third invisible object descended from a class named TConnector, itself descended from TVarSet. The purpose of this connector object is to provide a common boundary-value variable (or variables) to be used by the adjoining model components as part of their solution and to receive from those model components, in exchange, sufficient information to establish the validity of the boundary variable. This information generally takes the form of some quantity that must be conserved across the connection.

Take steady heat flow for example. One might imagine two thin heat-conducting rods connected in series between an isothermal source and an isothermal sink. Each rod is a separate model component and the connection in the middle is the current point of interest. The physical principle governing heat flow is this: Heat flows across the connection until the rod endpoint temperatures are equal. Simple. The Sage equivalent goes like this: Heat flow is an implicit variable managed by the connector object (TConnector instance) in the middle. Each rod maintains its own independent solution, producing an axial temperature distribution as a function of connector heat flow. When the user connects the two rod ends together, each rod receives a pointer to the connector object which it can call upon to read the current value of heat flow. Meanwhile, the connector object receives pointers to the appropriate endpoint-temperature variables within both rod components. The connector evaluates the difference of these two temperatures as the to-be-zeroed evaluation function associated its implicit heat flow variable.

So it goes with all the various types of information that can pass between model component boundaries. When it comes right down to it, there are surprisingly few types of information that pass across boundaries. Forces,

pressures, heat flows, fluid flows, electrical currents, just about does it.

Visually speaking, boundary connections appear as numbered arrows attached to model-component icons as shown in figure 1. The illusion is that connection is merely a matter of successive mouse clicks upon oppositely-oriented and similar-typed arrows. Arrows are shown either right-pointing or left-pointing emerging from the sides of the model components. This works well for one-dimensional model domains where the left-pointing arrows correspond to the negative endpoint boundary and the right-pointing arrows to the positive endpoint boundary. I have not yet had to deal with multi-dimensional solution domains where boundary connections might be made at odd angles — for example, connections representing chemical bonds. But such connections should be possible without fundamentally altering the scheme of things.

Grids

Numerical models often employ computational grids for the solution of finite-difference approximations to continuous differential equations. Grids are descended from TVarSet, but the variables owned by the grid are only those that apply to the grid as a whole, like node spacing for example. The *nodes* of the grid, also descended from TVarSet, contain the actual solved or *state* variables. So a grid is then an indexed collection of child nodes (similar to an array), each of which contains an indexed collection of variables. The grid itself has methods for operations that apply to the grid as a whole, such as averaging and integrating. The nodes have methods for operations applying to individual nodes such as state variable evaluation or finite-differencing.

The evaluation functions for the variables in the nodes carry out the detailed operations of whatever physical model the grid is supposed to be representing. And these operations generally depend on node position within the grid. For example, in a grid representing the gas within a heat-exchanger duct, a node variable named Rho, representing density, might evaluate itself at even spatial indices by interpolating from neighboring values. At odd spatial indices, it might evaluate itself implicitly according to a finite-difference version of the continuity equation $\frac{\partial \rho}{\partial t} + \frac{\partial \rho u}{\partial x} = 0$. The operations of spatial and time derivatives would, of course, involve state variables from neighboring nodes. This very example appears in figure 5. Note that there is not much programming work involved to implement a finite-difference equation

```
function GasSxtNodeRhoFimpl(AParent: TGasSxtNode): Extended; far;
begin
  with AParent, GetGrid, GetModel do begin
    Result:= ( DTime[Rho] + DSpace[RhoU] ) / C0.Val;
    end;
end;
```

Figure 5: The implicit evaluation function accessed by a grid state variable Rho (density) contained within a node of a two-dimensional (position-time) grid of a gas domain. The returned result is the degree by which the finite-difference continuity equation is unsatisfied. Variable RhoU (density $\times$ velocity) is another grid state variable. C0 is a normalization constant defined in the model component owning the grid. Built-in grid methods DTime and DSpace perform time and spatial finite-differencing.

once the grid structure is in place.

From a programmer point of view the use of these object-oriented computational grids is different than the traditional approach. The programmer is freed from dealing with the grid structure each time something new is to be modeled. Instead, all that structure is inherited from the proper ancestral grid class. The focus is instead on programming a number of individual evaluation functions for particular state variables, according to position within the grid. There is some overhead involved in setting up a finite-difference scheme this way, but it is generally of a routine nature. The big payoff is in code reliability and maintainability. And not to be taken lightly is the ability to use inheritance to create family trees of computational grids of ever increasing complexity. For example, there might be an ancestral grid for solving the finite-difference equations associated with compressible gas flow in ducts. Descendant grids might add refinements for particular turbulence-transition models, or detailed empirical heat-transfer or pressure-drop correlations for particular flow geometry.

Solving

A typical aggregate of model components, grids and connectors, might comprise perhaps thousands of individual variables, many of which are implicit

variables with values to be solved so as to zero their evaluation functions, which are generally nonlinear. So who solves them?

A nonlinear equation solver does. The strategy is iterative, based on a sequence of linear equations which locally approximate the nonlinear equation system. The coefficients of the linearized equations result from numerical partial derivatives of the implicit evaluation functions taken with respect to the model's implicit variables. The result is a sparse matrix, solved with a special sparse-matrix solver. The solution becomes a search direction along which to seek the nonlinear solution. This process repeats until the nonlinear model equations are all satisfied within some prescribed tolerance.

Mathematically, each iteration takes the form of solving the equation

$$\boldsymbol{J}\Delta V = -F$$

for the step ΔV, where $\boldsymbol{J}$ is the Jacobian (partial-derivative) matrix and F is the to-be-zeroed function vector. This is Newton's method. While Newton's method works well for *nearly* linear system functions, it can fail to converge for nonlinear ones, especially when the starting value for V is far from the final solution. In particular, discontinuities of implicit-variable initial values across connections have been observed to be a chief cause of non-convergence in Sage. The implicit function components corresponding to these discontinuities are of the form $F = V_+ - V_-$, where V_+ and V_- are the implicit variable values on either side of the connection. Newton's method tries to zero the discontinuity in one step, which tends to destabilize the solution. It is not too difficult to avoid this problem by relaxing variable discontinuities more slowly, which is what Sage's nonlinear solver does.

The idea is to replace F on the right-hand side of the above equation by ΔF, the desired change in F. For most components, ΔF_j is just $-F_j$, meaning that F_j is allowed to step all the way to zero. But for key function components (those of the form $F = V_+ - V_-$), ΔF_j is some smaller amount — small enough to avoid destabilizing the solution. The maximum allowable ΔF_j is something that is left to individual instances of certain classes of implicit variables to decide. I mention this innovation because it is one of the few things about Sage's nonlinear solution strategy that is not commonly found in the literature on such things.

Also worthy of discussion is the use of numerical differencing for partial-derivative calculations. In my opinion, this is all but essential for keeping the programming task manageable and reliable. And it is reasonably efficient. Starting with the current baseline values of all evaluation functions,

the implicit variables are stepped off one by one and partial derivatives $\frac{\partial F}{\partial V}$ computed as two-point differences of each evaluation function and its baseline value. But, generally, each evaluation function depends on only a few neighboring implicit variable values. To take advantage of this sparsity, the nonlinear equation solver maintains for each implicit variable a data structure known as the *Affected Function Collection*, or AFC for short. The AFC contains the indices of all implicit variables with evaluation functions affected by the given variable. The AVC is created in an initialization step by accessing the evaluation functions for each implicit variable one by one, until the function calls ultimately terminate in references to other implicit variables. Then, the index of the implicit variable containing the evaluation function is stored in the AFC of each terminally referenced implicit variable. So by use of the AFC, only affected evaluation functions need be evaluated when calculating partial derivatives $\frac{\partial F}{\partial V}$ for any given V. The evaluation mechanism previously discussed is also helpful. As $\frac{\partial F}{\partial V}$'s are computed for a given implicit variable, repeatedly-accessed dependent variables evaluate themselves only once, after which they simply return a stored value.

Optimizing

A model solved according to the user-supplied values of its input variables is useful, but not as useful as it could be. Most engineers also want to optimize their designs. Optimization is one of those words that is loosely applied to a wide range techniques, but in Sage it involves:

- A set of independent input variables, known as the optimized variables, to be automatically varied
- An arbitrary number of equality or inequality constraints to be satisfied
- An objective function to be minimized or maximized

This is technically known as a nonlinear programming problem and the task of implementing the nonlinear programming algorithm is the job of a nonlinear optimizer, yet another object class. The nonlinear optimizer resides at a higher level than the nonlinear solver because it calls upon the solver repeatedly to do its job.

The nonlinear optimizer in Sage employs a variation of Powell's sequential-quadratic-programming method [1] which locally approximates the nonlinear

optimization problem by a succession of quadratic sub-problems (quadratic objective function and linear constraints) each of which is readily solved. The idea is that by doing this often enough and searching along the direction from the current point to where the quadratic minimizer lies, one will eventually converge to the actual nonlinear minimizer — or rather *a* minimizer if there happens to be more than one. Powell's algorithm builds up its quadratic programming problems as it steps along by accumulating second derivative information about the objective function and constraints. It then turns to a separate quadratic optimizer for the sub-problem solution. In the Sage implementation, the quadratic optimizer is a convex method suited to Powell's method, reported by Goldfarb and Idnani [2].

Constraints and Objective Function

Formulating the optimization problem is done interactively at run time. The user actually creates constraints and the objective function as required, which are themselves streamable software objects that become part of the model structure. Both involve, at their core, string expressions supplied by the user interactively at run time. These string expressions are exactly like those employed by the special user-defined variables mentioned earlier and, indeed, may reference these variables. An acceptable expression is any string involving numerical constants, referenced variables and the usual operations such as $+, -, *, /$, with nested parenthesis and obeying the usual rules of syntax common to most programming languages. Dealing with such string expressions requires Sage to have a built in parser in order to convert them to an operation tree which, when traced-through in tree-order, carries out the actual computation. Parsing in Sage is very much like the compile and link steps of a typical programming language and takes place just prior to model solution or optimization automatically, or by menu command at any time.

Larger Issues

So what happens if we put all these ingredients together? Well, for one, we wind up with a big program. If Sage were a book it would be a 1000 page novel. Right now about 600 pages support the general architecture, including visual interface, while the remaining 400 pages pertain to specific model components. Maybe, one day, the proportions will be reversed as model

components become more numerous — which they surely will. Especially if the model-component concept catches on in the computational world at large.

What are the chances of this — of one day being able to draw upon a vast library of model components to simulate all sorts of things beyond stirling machines? Bicycles perhaps? That Boeing 747? The nervous system of a leech? I believe there is a trend in this direction — and not just in this article. Although, it would certainly help matters if operating systems and programming languages would evolve in directions more supportive of interchangeable model components. Right now, in Sage, model components are non-standardized objects residing in statically-linked executable modules rather than standardized objects residing in dynamically-linked library files. Another problem is — and I am speaking from experience here — it is not always immediately obvious how to break down a complex system into self-contained interchangeable model components of maximum usability. But in spite of the obstacles, I personally find the interchangeable-model-component vision of the future more appealing than, say, another generation of word processors or spreadsheets.

The Scope Thing

But enough of the future. Sage is up and running right now and doing useful work in the hands of a small but diverse band of faithful engineers. These engineers are behaving as expected and constantly changing their models. Even asking for new model components. And guess what? Sage model components are proving to be especially reliable, maintainable and extendible — though still not always perfect in modeling the underlying physics. The largest scope model yet simulated has been an equilibrium periodic cycle for a three-stage refrigerator with about a dozen or so interconnected top-level components — cylinder spaces, heat exchangers, valves, reservoirs — comprising about 2000 simultaneously solved implicit variables. Perhaps not that many plumbing components compared to a typical chemical plant but you can do a lot with 2000 variables. Especially if each one is relatively intelligent. The optimization performed for the largest-scope model yet simulated involved about thirty optimized variables — mostly heat-exchanger dimensions — subject to about a dozen or so constraints with the objective of minimizing electrical input power for a given amount of refrigeration power.

Execution time was fast enough that one or two optimization trials could be conducted in an hour or so. For scopes much larger than this, Sage may have to polish up its sparse matrix solver and nonlinear optimizer but seems to be fundamentally sound.

References

[1] M. J. D. Powell, *A Fast Algorithm for Nonlinearly Constrained Optimization Calculations*, in: *Lecture Notes in Mathematics*, 630, Numerical Analysis (Proc. Biennial Conf. at Dundee, 1977), Springer-Verlag (1978)

[2] D. Goldfarb, A. Idnani, *A Numerically Stable Dual Method for Solving Strictly Convex Quadratic Programs*, Mathematical Programing, **27**, pp. 1–33, (1983)