APPLICATION OF THE SPARK KERNEL

Edward F. Sowell[1], Michael A. Moshier[2]
[1]Department of Computer Science, California State University, Fullerton
Fullerton California, 92834 USA
sowell@fullerton.edu
[2]Department of Mathematics and Computer Science, Chapman University
Orange, California 92866 USA

ABSTRACT

In the mid 1980s the monolithic nature of building energy simulation programs led to proposals for development of so-called "kernel systems," i.e., software environments that would make available to developers basic software modules and a supporting framework that could be used to construct new building simulation software. One of the outcomes of the ensuing work was the Simulation Problem Analysis and Research Kernel (SPARK). Although the current SPARK release can be viewed as a limited realization of the kernel idea, it falls short in that the internal methods can only be accessed within the context of a SPARK executive. This paper discusses two new ways in which the SPARK internal methods can be employed by model developers, leading to a fuller realization of the kernel system idea. First, a new facility called *SPARK Model Functions* is described that allows the SPARK internal methods to be used to create subsystem models of arbitrary size and complexity that can be called by foreign executive programs. Second, a new feature called *Multivalued Objects* allows easy and efficient integration of legacy models written in procedural languages into SPARK models. Together, these new features provide an Application Programmer's Interface (API) that better exposes "the K in SPARK" to the software developer.

BACKGROUND

In 1985 a meeting was convened at the Lawrence Berkeley National Laboratory (The Berkeley Lab) to consider the state of the building energy system simulation art, with the intent of establishing a direction toward a new generation of such software. Proposals advanced there held that what was needed was not development of another, all encompassing computer program, but rather a collection of the body of essential, software-expressed, building modeling technology in a framework that could be used by any software developer to easily assemble new programs to meet both general and specific building simulation needs. This was referred to as the Energy Kernel System (EKS) (Clarke 1986). Subsequently the UK sponsored development of a prototypical EKS, the outcome of which was summarized by Clarke at the IBPSA Building Simulation '93 Conference (Clarke and MacRandal 1993). In the US a parallel effort, initially called the US/EKS, resulted in a prototypical software system called SPANK (Simulation Problem ANalysis Kernel) (Sowell, Buhl et al. 1986).

Although both the UK EKS and SPANK emerged from the same Energy Kernel idea, these efforts took radically different approaches. The UK EKS can be characterized as object oriented with the objects typically being large procedural modules stored in a database and assembled by the user, along with a solver. Internally these models are expressed in vector-matrix form and the solver incorporates sparse techniques for efficient solution. The US approach, which is now known as SPARK (Simulation Problem ANalysis and Research Kernel), works at a lower level, with objects representing individual equations assembled into a network representing the entire problem.[1] The problem network is mapped onto a mathematical graph internally, and graph algorithms are used to automatically determine an efficient solution sequence. Working directly with individual equations has several attractions, including accommodation of an input/output free, non-algorithmic modeling paradigm. Input/output free means that variables which are to be inputs and those which are to be calculated need not be specified *a priori*. Non-algorithmic (also know as declarative) programming means that the solution sequence is determined automatically rather than being specified by the modeler. The progress of these two different Kernel Systems has been reported regularly in the IBPSA conference series and elsewhere. The most up-to-date information on the UK project is available at http://www.bitd.clrc.ac.uk/Activity/ACTIVITY=EKS while that of SPARK is available at http://simulationresearch.lbl.gov/.

In the current release, SPARK can be described as a stand-alone program comprising a solver engine and a user interface for describing the physical system to be simulated. Three choices of user interface are available: the text based SPARK network description

[1] Since its objects do not support inheritance, SPARK is more properly called "object-based" rather than object oriented.

language, and two different graphical user interfaces, namely Ayres Sowell Associates (ASA) *WinSPARK*, and the Berkeley Lab *Visual SPARK*. By any of these interfaces users can describe physical systems for simulation with a great deal more flexibility than is possible with monolithic, whole-building simulators, and solve them with more sophisticated symbolic and numerical techniques than are available in subroutine based modular simulators such as TRNSYS (anon. 2000) or HVACSIM+ (Park, Clark et al. 1985). Thus described, SPARK is an application program, rather than a kernel system in the original sense. Nonetheless, a software developer with a new notion of how simulation models should be described can translate that notion into a SPARK input language file (or into an alternative, lower level problem description file) and then use the SPARK input and problem *setup* processor to generate an executable solver. In this limited sense the release can also be viewed as a realization of the kernel idea.

While the Kernel System idea continues to be appealing, it is hard to argue that it has been widely accepted. Although some of the technology developed in the UK EKS project no doubt has found its way into other software systems such as the *ESPr* program (Clarke 2002), the EKS itself has not yet been taken beyond the prototype stage (MacRandal 2002). And while SPARK has a small user community one doubts that it has drawn many away from the large communities using traditional modular programs such as TRNSYS, and as yet has not been integrated into any of the whole building programs as originally hoped.[2] At the same time, we continue to see reports of new building simulation programs that seem to start fresh in the representation and solution of HVAC system models, apparently oblivious to the EKS efforts. Assuming that the basic premise of kernel system was and is correct, one cannot help but wonder why this situation exists. It can be explained in part by the natural creative urges on the part of building simulation theorists and software developers. But even so, one wonders if these developers could not have expressed their creativity in the overall architecture and interfaces of their new applications, using the kernel system technology at some (unseen and uninteresting) lower level. Since, apparently, this has not often happened, it must be that either the implementation of the kernel system projects on both sides of the Atlantic were somehow faulted, or they have been ill presented. If the former is the case, little can be done. But if the latter is the more correct assessment, it could be that if the EKS and SPARK efforts had focused more on the Kernel, and simple programmer's interfaces to it, rather than on "all or none" software systems, other developers might have used it to augment their own creative projects. We believe that this is the case, and that the problem can be corrected. That is, hope remains for success of the energy kernel idea, not in stand-alone programs but in Application Programmer's Interfaces (APIs) to the underlying kernel system functionality.

Thus in order to more fully realize the original Energy Kernel System objectives, we seek to make the SPARK internal methods directly accessible to model developers. For one thing, we want developers to be able to use these internal methods to create system models, of arbitrary size and complexity, which can be called by foreign executive programs. This would allow SPARK models to be to be used in the context of other simulation environments, and in situations where the SPARK executive does not accommodate special simulation needs. For discussion purposes here we shall refer to this new capability as *SPARK Model Functions (SMF).*

Situations also arise where a developer or analyst wishes to use a model expressed in an algorithmic language *within* a SPARK model. This comes up where there is an existing, trusted model (sometimes called *legacy code*) written in a procedural language, e.g., *FORTRAN*, *C*, or *C++*, and time or other factors argue against re-implementation as an equation-based SPARK macro class. Or, there may be small sets of equations within a system that are numerically problematic for a global solver, but which can be reliably solved simultaneously with well-known procedural algorithms. In both of these situations there are multiple equations being solved for multiple variables simultaneously within the subsystem model. This is in contrast to the normal SPARK policy of breaking subsystems (macro objects) into the constituent individual equations and variables to be solved globally. To better accommodate such subsystem models, there is a need for SPARK to accept subsystem models that provide *multiple values* back to the global solver, rather than the normal single valued atomic objects. For discussion purposes here we shall refer to this new capability as *Multi-Value Objects (MVOs).*

We begin with a brief explanation of SPARK internals. This is followed by an overview of the SPARK modifications needed to implement *SMFs* and *MVOs*, along with some example usage.

SPARK INTERNAL STRUCTURE AND OPERATION

A SPARK problem is defined in terms of linked objects using a textual network description language, typically saved in a *probName.pr* file. This description can be hierarchical, using macro objects and macro ports.

As shown in Figure 1, the first processing step on a SPARK problem file is parsing. The parser reads and parses the *probName.pr* file and all class object files referred to in the problem file, descending recursively

[2] Integration of SPARK with Energy Plus is planned for release by The Berkeley Lab.

to parse all macro objects found in that file and its dependencies. The output of the parser is the SPARK setup file *probName.stp*. This is another textual representation of the problem, but one in which all macro objects and links have been resolved into their atomic parts. That is, this is a "flat" problem description rather than hierarchical, composed entirely of atomic objects, each representing a single equation. Links between these atomic objects represent individual variables. In both the original problem description (*probName.p*r) and the flat description (*probName.stp)* links representing variables with user specified values are marked as *inputs*, comprising the input set; other links are the problem unknowns that must be solved for, i.e., the output set.

The next step, carried out by the SPARK *setup* program, applies a series of graph-theoretic analyses to determine the structure of the problem graph in terms of strongly connected components, cycle cut sets within these components, and topological orderings. This establishes the order in which the graph components must be processed during numerical solution, and the visitation order of the nodes within each component, leading to an efficient overall problem computation sequence. Efficiency derives from the reduction in size of simultaneous solution sets.

A brief description of internal problem representation in terms of *C++* objects is necessary for better understanding the extensions needed for *SMFs* and *MVOs*. The computation sequence, as well as data needed to support it, is expressed as a collection of *C++* object instantiations and initializations. The *C++* source language for these instantiations and initializations are developed by the *setup* program and saved in a file named *probName.cpp*. The most important object in this file is an instance of a *C++* class *TSPARKProblem* called *myProblem* that embeds the entire problem solution sequence and data. The numerical routines operate entirely on this object to determine the output set from the given input set.[3]

An important feature of SPARK is the non-directionality of atomic (and macro) objects. That is, there is no *a priori* designation of input and output variables when a SPARK class is defined. This is achieved by providing every atomic class with a set of inverse functions so that, ideally, the object equation could be used to solve for any single variable appearing in the class equation. As part of the problem definition the user identifies problem inputs, and this allows the *setup* program to identify a single variable to be solved for, and the corresponding needed inverse function, at each object. The aforementioned computation sequence is in fact an array of pointers to these inverse functions, and evaluation of the entire component is a matter of stepping through this array and executing these functions one by one. If the component being processed has cycles the array is traversed repeatedly until convergence. In order for this to work the problem solver build process must be provided with compiled code for each of these inverse functions. Therefore s*etup* also provides a list of the source files for these inverse functions.

Once *setup* has constructed the *probName.cpp* file, as well as the list of source files for the required SPARK inverse functions, all these files can be compiled for linkage with the SPARK solver fixed code, producing an executable solver program. The user executes the solver program to get numerical answers.

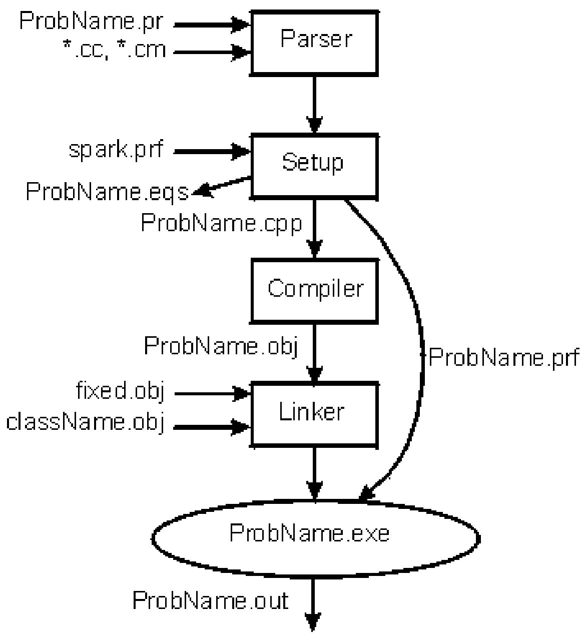

Figure 1 SPARK processing diagram.

The *main()* function of solver executable is the result of compilation of a fixed main program that has the basic structure shown below:

```
int main()
{
        …
        // Initializations
        myProblem.initialSolve();
        myProblem.Solve();
        // Final cleanup etc.
        return 0;
}
```

[3] In the case of time varying inputs and/or dynamic elements in the problem, this solution is done at discrete times over the specified solution interval. However, if the intention is to export a *SMF* for use by a foreign executive that handles time advancement, this aspect of SPARK processing can be ignored.

The *myProblem* object, which completely defines the model to be solved, is instantiated *globally* in the *probName.cpp* file, which is separately compiled and linked with this fixed main program. The *initialSolve()* member function of the *TSPARKProblem* class carries out the numerical solution of the problem, component by component, at the specified initial conditions and generates requested reports. The *Solve()* member function does much the same, but includes a loop in which time is advanced for dynamic simulations. Both of these functions call a lower level function *Evaluate()* that actually carries out the previously found solution sequence. (The principle difference between *InitialSolve()* and *Evaluate()* is that the former does data input prior to calling *Evaluate()*.)

SPARK MODEL FUNCTIONS

Starting with the basic ideas explained above, there are several ways *SPARK Model Functions* might be implemented. One way, implemented and described by Curtil (Curtil 2002), provides an API consisting of calling conventions for the needed SPARK internal methods. Access to these methods enables a developer to create *multiple* SPARK *problems* (in the meaning discussed in the previous section) *within a single SPARK executable*, and to write a *customized executive routine*, replacing the standard SPARK *main()* function. This greatly extends flexibility in that each problem can be considered, in effect, as a *SMF* even though they are not actual functions. These can be executed in any order and sequence, as determined by the developer's customized executive. However, the approach does *not* allow the developer to export functions in a form that can be linked into completely different software. That is, one can only use Curtil's *SMFs* within the SPARK framework.

Another way to implement *SMFs* requires a few changes to existing SPARK fixed code, but allows automatic generation of *SMFs* exportable as ordinary *C++* functions. With this approach, as with Curtil's, the SPARK input language is used to express a model of the system for which a solver function is wanted, say *mySystem*. The standard *parser* and a slightly modified *setup* programs are executed in the usual manner to get the *probName.cpp* file and list of needed inverses. Instead of compiling and linking with *main()*, however, we compile a function called *mySystemSM()* (where SM stands for System Model), also generated by *setup*, that initializes system model inputs from an argument input array and returns the results to an argument output array. For example, if our "system" were simply a mixer blending two moist air streams the function would be (approximately):[4]

```
int  mixerSM(const double * in, double*
out)
{

       // initializations
       ...
       // Set the in arguments
       myProblem.SetValue("mEnt1", in[0]);
       myProblem.SetValue("TEnt1", in[1]);
       myProblem.SetValue("wEnt1", in[2]);
       myProblem.SetValue("mEnt2", in[3]);
       myProblem.SetValue("TEnt2", in[4]);
       myProblem.SetValue("wEnt2", in[5]);

       myProblem.Evaluate(); // Solve
       // Retrieve the results
       out[0] =
myProblem.GetValue("mLvg");
       out[1] =
myProblem.GetValue("TLvg");
       out[2] =
myProblem.GetValue("wLvg");
       out[3] =
myProblem.GetValue("hLvg");
       out[4] =
myProblem.GetValue("hEnt1");
       out[5] =
myProblem.GetValue("hEnt2");
       // Clean-up etc
       ...
       return 0;
}
```

Note that *myProblem* is the name of a *TSPARKProblem* object representing the system internally. We make use of the *TSPARKProblem GetValue()* and *SetValue()* class member functions to transfer argument values to and from *myProblem*, and use the *Evaluate()* member function to solve it. This *mixerSM()* function can then be compiled, along with the atomic mixer object inverse *mixer.cpp* and linked into a user library. The developer's application program can then be linked with this library, allowing the *mixerSM()* function to be called as needed. Although this is a simple example, a system of any size and complexity could be handled in the same manner.

The limitation of the approach as described above is that in standard SPARK the object named *myProblem* is globally defined, meaning that *only one* system model can be used in a particular simulation. That is, you could not use both a *mixerSM* and a *collectorSM* in a single application because there can be only one *myProblem*.[5]

To get around this limitation we must make a few small changes to the standard SPARK *probName.cpp* file. The needed changes can be made either by

[4] Typically, a *SMF* would be much more complex, involving many objects rather than only one.

[5] This is because SPARK was originally designed to solve a single problem at a time.

modifying the *setup* program, or by post processing the *probName.cpp* file.

The ability to have more than one instance of a *TSPARKProblem* in a single simulation is achieved with the *namespace* feature of *C++*. This feature allows the programmer to define the scope of identifiers. For example, suppose we write:

```
namespace MixerSM{
        TSPARKProblem myProblem;
…
}; // end MixerSM namespace
namespace CollectorSM{
        TSPARKProblem myProblem;
…
}; // end CollectorSM namespace
```

The *C++* compiler can then distinguish between the two usages of *myProblem*, so that two separate objects are instantiated. The scope resolution operator :: is used to distinguish them. The *mixerSM()* function is then:

```
int  mixerSM(const double * in, double*
out)
        ...
        actvPrb = &MixerSM::myProblem;
        actvPrb->Initialize( &MyRTControls
);
        // Set the inputs
        actvPrb->SetValue("mEnt1", in[0]);
        actvPrb->SetValue("TEnt1", in[1]);
        actvPrb->SetValue("wEnt1", in[2]);
        actvPrb->SetValue("mEnt2", in[3]);
        actvPrb->SetValue("TEnt2", in[4]);
        actvPrb->SetValue("wEnt2", in[5]);
        // Solve
        actvPrb->Evaluate();
        // Retrieve the results
        out[0] = actvPrb->GetValue("mLvg");
        out[1] = actvPrb->GetValue("TLvg");
        out[2] = actvPrb->GetValue("wLvg");
        out[3] = actvPrb->GetValue("hLvg");
        return 0;
} // end of mixerSM
```

Note that we assign the address of the *MixeSMr::myProblem* object to a global *TSPARKProblem* pointer so it can be accessed in other SPARK internal functions as well as in the *mixerSM()* function.

With this technique we can have any number of *SPARK Model Functions* active in an application program. We envision simulation software developers using SPARK to implement component and subsystem models in this manner, and compiling and them into a dynamically linked library (DLL) for use in their own over-all system models, thereby gaining the advantages of SPARK's advanced model description and solution techniques.

THE WINSPARK MODEL FUNCTION AND LIBRARY GENERATORS

In this work the above idea was implemented by modification of the SPARK *setup* program. The *probName.cpp* file now contains the *probNameSM()* function and a test driver *main()* that demonstrates usage, in addition to the normal code. This code is marked for conditional compilation so it can be omitted when building a conventional SPARK problem. A separate program called *sparkMFG* (SPARK Model Function Generator) compiles *probName.cpp* with flags set so as to produce *probNameSM.obj* and the test driver *probNameSMDrv.exe*. The *SparkMFG* program can be executed from the command line:

```
C:\myProject\mixer> SparkMFG mixer <enter>
```

Or, the operation can be carried out within the *WinSPARK* environment using the *Generate model as function* choice on the Run command menu, Figure 2.

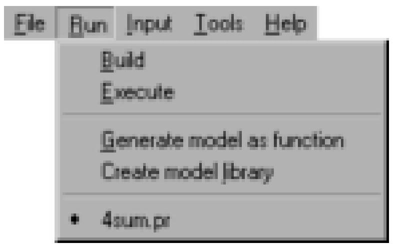

Figure 2 WinSPARK Run menu

Also added to the Run menu is the choice *Create model library*. When this is selected all system model functions that have been generated for the current project are compiled and linked into a Dynamic Link Library (DLL)., which can be compiled and linked (along with the SPARK solver DLL) into the developer's application.

EXAMPLE USAGE

As an example of the power of the above technique the modified *WinSPARK* has been used to develop a demonstration HVAC Toolkit (Brandemuehl 1993; Sowell and Moshier 1995) that can be used in a Microsoft *Excel* worksheet, using the facility for calling DLL functions from Visual Basic for Application (*VBA*). We describe the implementation only in outline to save space.

First, a problem is defined in a SPARK project for each toolkit component to be included, e.g., mixer, collector, heating coil, cooling coil, zone, etc. The *SparkMFG* program (or the Generate Model as Function menu choice) is applied to each problem individually in order to test the functions. When all are judged satisfactory the Create model library menu choice is used to create a DLL containing all of the functions.

If the intent is to use the SPARK Model Functions in a non-C++ environment they will have to be "wrapped" in a module for the target environment, observing the foreign calling conventions. For example, if we intend to use the functions in Excel, we wrap them in *VBA* functions that take input arguments from, and write results to, worksheet cells. The one for the mixer shown below typifies these *VBA* functions. Note that we simply load the SPARK-generated *mixerSM()* function's input arguments with the cell addresses where these values reside, call the *mixerSM()* function, and then transfer the function output arguments to their respective cells:

```
    Sub MixerDriver()
        Dim inArgs(5) As Double, outArgs(5)
As Double
        Dim ec

SetInputs("Mixer", inArgs(0),
{"mAirEnt1","TAirEnt1","wAirEnt1","mAirEnt2
","TAirEnt2","wAirEnt2"})
ec = mixerSM(inArgs(0), outArgs(0))
        If ec = 0 Then
SetOutputs("Mixer", outArgs(0),
{"mAirLvg","TAirLvg", "wAirLvg",
"hAirLvg", "hAirEnt1", "hAirEnt2"})
        End If
    End Sub
```

Here, *SetInputs()* and *SetOutputs()* are *VBA* functions that use look-up to find the names and set up corresponding cell references. The function can then be invoked as would any *VBA* function. For example, an entire system can be simulated by another *VBA* function that calls the individual component models:

```
Sub simulate()
        Call CcsimDriver
        Call ZonePropcont
        Call DrcctrDriver
        Call ZoneDriver
        Call DivsimDriver
        Call MixerDriver
        Call EnthalpyDriverDriver
End Sub
```

Let us be clear that this example is probably not the most efficient way to model HVAC systems. For one thing, developing the *VBA* functions and interconnections between the several models is a tedious, error prone task. A more serious limitation is lack of automatic iteration to convergence. However, given the large community of spreadsheet analysts (even in the building simulation field), this may be an attractive application in spite of these limitations. If one were to continue this line of development, leading to a truly useful toolkit for *Excel*-based HVAC simulation, the *VBA* functions should be generated automatically, and the *simulate()* function should embed a global solver, either using the *Excel* nonlinear solver or perhaps the SPARK solver in suitable form. Here, we present the idea primarily as an indication of how *SMFs* can be used to develop system simulations outside of the SPARK environment.

MULTI-VALUED OBJECTS

As noted earlier, normal SPARK atomic objects represent a single equation and therefore can produce only a single value in a model. This is the preferred basis for modeling in SPARK because it exposes all equations and variables to the SPARK graph algorithms, thus allowing optimal solving. Also, it allows greater modeling freedom since designation of input and output variables can be deferred until solution time.

In spite of the advantages of equation based modeling, sometimes users have reason to solve certain parts of the problem with normal, algorithmic functions. For example, one may have a complex model implemented in a procedural language such as *C* or *FORTRAN*, but insufficient time to extract the underlying mathematical model and reimplement it as a SPARK, equation based, model. Another example is a subproblem that is not well suited for iterative numerical solution, but can be solved symbolically without too much difficulty. The classic example of this in an HVAC context is zone temperature control, which in the most basic form has two equations, one a quadratic and the other a piecewise linear function, Figure 3. It is well known that iterative solution can fail if the slope of the piecewise linear function is too steep where the two functions intersect. However, one can easily construct an efficient algorithm by symbolically solving for the intersection of the quadratic with each line of the piecewise linear function (or its extension), then picking as the correct solution the point that lies on a legitimate segment of the piecewise linear function .[6]

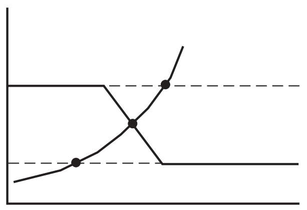

Figure 3 Quadratic and piecewise linear solution

In cases like this the solution function produces more than one output. For example, the function *solveMyModel(x, y)* for the above example produces both *x* and *y*. To use this multivalued function in a SPARK problem you can write two atomic classes, each having an inverse function that calls the

[6] The algorithm is given in the example *qpwl* in the WinSPARK documentation.

solveMyModel(x, y) function. One of these atomic classes would be designed to return the *x* result of the call to *solveMyModel()*, another to return *y*. The two atomic classes could be wrapped in a single macro class, providing a single modeling entity that represents the subproblem.

The above strategy has been available and usefully employed from the earliest days of SPARK. However, the obvious disadvantage is that at run time the *solveMyModel(x, y)* function gets executed twice: once to return *x*, and once to return *y*. For some situations this may not be a terrible loss, but if the algorithm is time consuming and the number of outputs is large the problem run time may suffer significantly.

Recently, ASA has extended SPARK to offer a more efficient way to handle multivalued objects like this. The approach used is very similar to that described above. In fact, *MVOs* are modeled exactly as described above, but at run time the solution process is monitored so that *all calls but the first* to the multivalued function can be skipped. For example, if *y* is the first needed result when the problem is solved, *solveMyModel(x, y)* is called when the object producing *y* is calculated, but not when *x* is needed. (This approach was first described by Buhl and Sowell (Buhl, Erdem et al. 1993).) To use this new *MVO* facility a *MVO* class, the multivalued function, and the atomic class inverse function must follow a special protocol described in the *WinSPARK* documentation (Sowell 2001). Much of the code in a *MVO* inverse function is fixed, meaning that you can easily construct them with a "cut and paste" process. Or, you can use the *MVO* choice on the *WinSPARK* Symbolic menu to construct your *MVO* class automatically once you have created the multivalued solve function. Examples, including the one shown in Figure 3, are included in the *WinSPARK* documentation. These examples have solve functions written in *C/C++*, but in principle they could be expressed in any language for which you have a compiler that produces object-level compatibility with Microsoft Visual *C/C++*.

DISCUSSION

The two extensions to SPARK described herein, *SMF* and *MVO*, are currently available in *WinSPARK* and may be included in some form in later SPARK releases from The Berkeley Lab. Taken together, they provide a useful Application Programmer's Interface (API) that makes the SPARK methodology available to simulation system developers that choose to work in non-SPARK environments. Examples might include:

(1) Porting existing model libraries into spreadsheet functions, thus providing high quality models to analysts who prefer that environment,
(2) Providing user-defined secondary and primary systems for whole-building programs such as DOE-2 and Energy Plus,
(3) Generating models for component-based programs such as TRNSYS and HVACSIM+; and
(4) Providing equipment manufacturers a straightforward means for developing high quality models of their products for use in in-house simulation tools.

We believe that this new capability represents a more complete and useful realization of the original Energy Kernel concept. At the same time, it must be recognized that in both *SMF* and *MVO* we are departing from the pure, equation based, non-procedural basis of the SPARK methodology, and this means that some of the SPARK advantages will not be realized. For example, if two parts of a system are modeled as separate *SMFs* and brought back together in a different modeling environment the overall solution may not be as efficient as could have been realized if both parts were modeled together as a single SPARK problem. Moreover, a *SMF* always represents a single input/output combination, while many such input/output combinations are possible for the underlying SPARK problem. That is, the input/output free nature is lost in the process of generating the *SMF*.[7] These disadvantages are most evident in the *Excel* HVAC toolkit example, where the *SMFs* are individual SPARK component models, e.g., mixers, collectors and cooling coils. In many cases these models are so small that little runtime speed advantage is being gained by the graph-theoretic analysis, yet the overhead of doing it remains. Therefore this particular usage must be justified entirely by the convenience of modeling in the spreadsheet environment. In general, it is likely true that the best usage will be characterized by placing as much of the problem as possible on "the SPARK side," so to speak, when these new features are used to integrate SPARK into other applications. That said, we nonetheless encourage software developers to take advantage of this facility as they see fit, believing that they will gain some of the advantages of the advanced modeling and solving techniques built into SPARK, while at the same time retaining overall control of the architecture of their own applications.

CONCLUSIONS

We have attempted to review the state of the Energy Kernel System idea as represented in the UK EKS project and the Berkeley Lab/ Ayres Sowell SPARK project. The full benefit of the kernel idea appears not

[7] However, this disadvantage is not severely limiting because it is an easy matter to change the input designations in the SPARK model and regenerate the *SMF*.

to have been realized, as we continue to see building energy tools being developed without taking advantage of the EKS or SPARK technology. We believe that this is due in part to the lack of suitable Application Programmers Interfaces. Addressing this issue, we presented an implementation of a two-part API for *WinSPARK*. One part allows simulation software developers to automatically generate *SPARK Model Functions*, which are *C++* functions representing SPARK problems. Since these are callable from any compatible language, e.g., *VBA* or *FORTRAN*, this feature means that developers can take advantage of SPARK's advanced modeling and solution methods within the context of their own applications. The second part of the API, called Multivalue Objects goes in the opposite direction, i.e., it allows foreign code to be efficiently incorporated in SPARK models. This provides a migration path for developers with a significant body of legacy code to rapidly port their models to the SPARK environment. We believe that features of this nature will encourage the flow of kernel technology into mainstream building simulation.

ACKNOWLEDGEMENTS

This work has been sponsored in part by the Assistant Secretary for Conservation and Renewable Energy, Office of Building Technologies, Building Systems and Material Division of the U. S. Department of Energy.

REFERENCES

anon. (2000). TRNSYS 15: A Transient System Simulation Program. Madison, WI, Solar Energy Laboratory, University of Wisconsin-Madison

Brandemuehl, M. J. (1993). HVAC 2 Toolkit: A Toolkit for Secondary HVAC System Energy Calculations. Boulder, Colorado, Joint Center for Energy Management, University of Colorado

Buhl, W. F., A. E. Erdem, et al. (1993). Recent Improvements in SPARK: Strong Component Decomposition, Multivalued Objects, and Graphical Interface. *Building Simulation '93*, Adelaide, International Building Performance Simulation Association. 283-90.

Clarke, J. A. (1986). The Energy Kernel System: A Technical Overview. *Proceedings of Second International Conference on System Simulation in Buildings*, Liege, Belgium, Univ. of Liege, Thermodynamics Laboratory.

Clarke, J. A. (2002). ESP-r, Strathclyde University. **2003** http://www.esru.strath.ac.uk/Programs/ESP-r.htm

Clarke, J. A. and D. F. MacRandal (1993). The Energy Kernel System: Form and Content. *Building Simulation '93*, Adelaide, International Building Performance Simulation Association. 307-315.

Curtil, D. (2002). SPARK Problem Driver API. Berkeley, Simulation Research Group, Lawrence Berkeley National Laboratory http://simulationresearch.lbl.gov/

MacRandal, D. (2002). The Energy Kernel System. **2003** http://www.bitd.clrc.ac.uk/Activity/ACTIVITY=EKS

Park, C., D. R. Clark, et al. (1985). An Overview of HVACSIM+, a Dynamic Building/HVAC Control Systems Simulation Program. *Proceedings of the First Building Energy Simulation Conference, Dec. 3-6.*, Seattle, WA, International Building Performance Simulation Association.

Sowell, E. F. (2001). Multivalued Objects in SPARK, Ayres Sowell Associates, Inc. **2003** http://www.ayressowell.com/WinSPARK/readmeMVO.txt

Sowell, E. F., W. F. Buhl, et al. (1986). A Prototype Object-based System for HVAC Simulation. *Proceedings of the Second International Conference on System Simulation in Buildings*, Liege, Belgium, Univ. of Liege.

Sowell, E. F. and M. A. Moshier (1995). HVAC Component Model Libraries for Equation-based Solvers. *Building Simulation '95*, Madison, WI, International Building Performance Simulation Association.