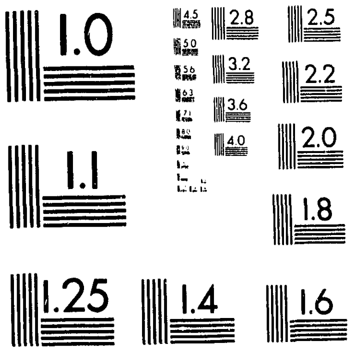
Centimeter
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 mm
1 2 3 4 5
Inches


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3

LBL-32815

Automatic Code Generation in SPARK: Applications of Computer Algebra and Compiler-Compilers

Jean-Michel Nataf* and Frederick Winkelmann

Simulation Research Group
Building Technologies Program
Energy and Environment Division
Lawrence Berkeley Laboratory
Berkeley, CA 94720

September 1992

Abstract

We show how computer algebra and and compiler-compilers are used for automatic code generation in the Simulation Problem Analysis and Research Kernel (SPARK), an object-oriented environment for modeling complex physical systems that can be described by differential-algebraic equations. After a brief overview of SPARK, we describe the use of computer algebra in SPARK's symbolic interface, which generates solution code for equations that are entered in symbolic form. We also describe how the Lex/Yacc compiler-compiler is used to achieve important extensions to the SPARK simulation language, including parametrized macro objects and steady-state resetting of a dynamic simulation. The application of these methods to solving the partial differential equations for two-dimensional heat flow is illustrated.

1. Introduction

The Simulation Problem Analysis and Research Kernel (SPARK) is a new equation-based, object-oriented simulation environment for modeling complex physical systems. SPARK takes algebraic equations as elementary objects and creates simulation programs for virtually any combination of the equations. SPARK is being developed for the U.S. Department of Energy by Lawrence Berkeley Laboratory and California State University at Fullerton.

This paper describes how SPARK automatically generates code using symbolic manipulation and computer algebra. Section 2 gives a brief overview of the SPARK environment. In Section 3 we describe the computer algebra tools that relieve the SPARK user of most of the tedium of object and module creation, to the point where simply specifying the equations and their interconnections is enough to generate a flexible simulation program. The use of compiler-compilers for code generation is discussed in Section 4. Finally, we show an example application in Section 5.

* Now at Groupe Informatique et Sciences Energetiques, ENPC, La Courtine-Cedex, Noisy le Grand, France.

This work was supported by the Assistant Secretary for Conservation and Renewable Energy, Office of Building Technologies, Building Systems and Materials Division of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098.

2. The SPARK Environment

SPARK generates a solution procedure that is tailored to each particular simulation problem, then implements that procedure in a program that it automatically generates in the C language. The overall organization of SPARK is shown in Fig. 1 [Buhl 1990].

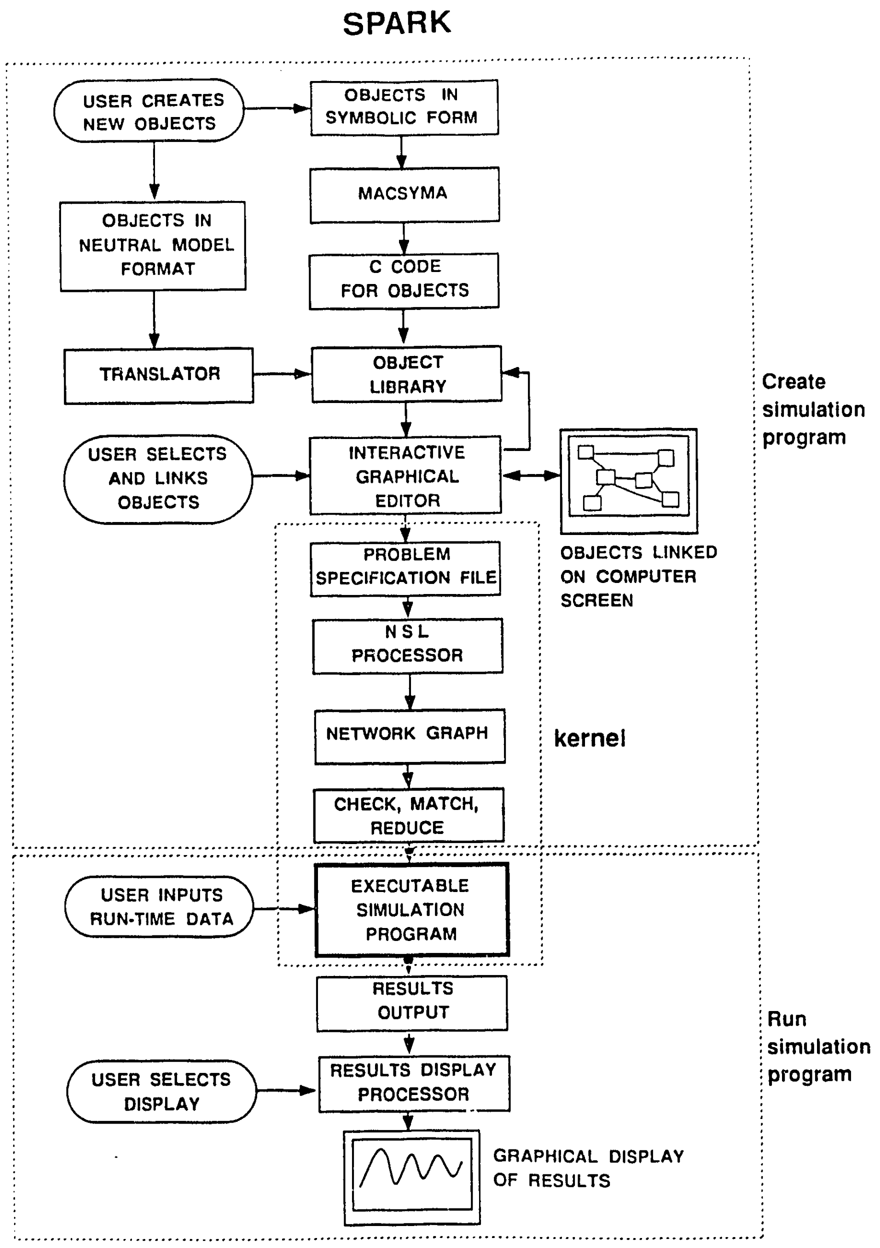

SPARK
USER CREATES NEW OBJECTS
OBJECTS IN SYMBOLIC FORM
MACSYMA
OBJECTS IN NEUTRAL MODEL FORMAT
C CODE FOR OBJECTS
TRANSLATOR
OBJECT LIBRARY
USER SELECTS AND LINKS OBJECTS
INTERACTIVE GRAPHICAL EDITOR
OBJECTS LINKED ON COMPUTER SCREEN
Create simulation program
PROBLEM SPECIFICATION FILE
N S L PROCESSOR
NETWORK GRAPH
kernel
CHECK, MATCH, REDUCE
USER INPUTS RUN-TIME DATA
EXECUTABLE SIMULATION PROGRAM
RESULTS OUTPUT
Run simulation program
USER SELECTS DISPLAY
RESULTS DISPLAY PROCESSOR
GRAPHICAL DISPLAY OF RESULTS

Figure 1: Configuration of the Simulation Problem Analysis and Research Kernel (SPARK). Shaded boxes are programs; unshaded boxes are files. Ovals show user actions.

The user interacts with SPARK in four basic ways: (1) defining objects (which represent the equations of a physical system), (2) linking objects together to define the simulation problem to be solved, (3) specifying run-time data (parameters and time-varying input data); and (4) specifying desired output. The objects are defined in text files, either as mathematical equations or as component models in Neutral Model Format [Sowell 1989]. These files are processed symbolically with programs written in MACSYMA [MIT 1983], producing C language functions and objects that are stored in libraries. Problems are defined by interconnecting objects using the graphical user interface, producing a problem specification file in the Network Specification Language (NSL)[Anderson 1986]. From the NSL description, SPARK generates internal data structures based on graphs. Matching and reduction algorithms are used with these graphs to automatically devise an efficient solution algorithm, producing an executable program for each particular problem. This program reads constant and time-varying input data from files, producing the problem solution. The output processor reads the results file and generates graphical displays according to interactive user requests.

The initial version of SPARK (called SPANK — Simulation Problem Analysis Kernel [Anderson 1986]) handled only steady state problems, i.e., those involving nonlinear equation systems without time derivatives. SPARK was extended to dynamic systems in 1989, and can now handle problems involving nonlinear equations with time derivatives on any of the variables.

SPARK has been successfully used for solving problems encountered in building energy simulation, including air conditioning systems [Buhl 1990], desiccant dehumidification [Nataf 1991], lighting systems [Sowell 1990], and coupled natural convection and conduction [Buhl 1990].

3. The Symbolic Interface to SPARK

The objects in SPARK are equations whose interfaces with the outside (and with other equations) are the equation variables (see Fig. 2). Linking two objects means that one or more variables are shared by the equations, as illustrated in Fig. 3. Thus, SPARK needs to be told what the interfaces of each equation are, how they are linked with the interfaces of the other equations, and under what name. This information is supplied in "object files" that encapsulate all information about each equation. Object files can be linked together to make macro object files (which are equivalent to systems of equations).

The SPARK solution method requires that the user provide functions, in the C language, that solve each equation in terms of each of its variables. For an equation of the form

$$f(x,y,z,\cdots)=0,$$

this means that the inverse functions g, h, etc., have to be specified, such that

$$x=g(y,z,\cdots),\ y=h(x,z,\cdots),\ \text{etc.}$$

Although SPARK users can generate object files and function files by hand, the process is tedious, error prone and time consuming. For example, for an equation with N explicit variables, there are, in general, N C functions to supply, plus one SPARK object file that tells which functions are associated with which variables. An object corresponding to a physical process or component is usually described by several equations, each of them having an associated object file and retinue of function files.

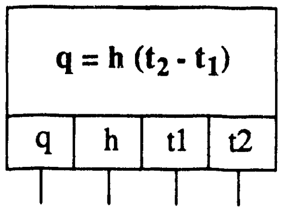


Figure 2: An elementary SPARK object, which represents a single equation. The interfaces of the object are the equation variables.

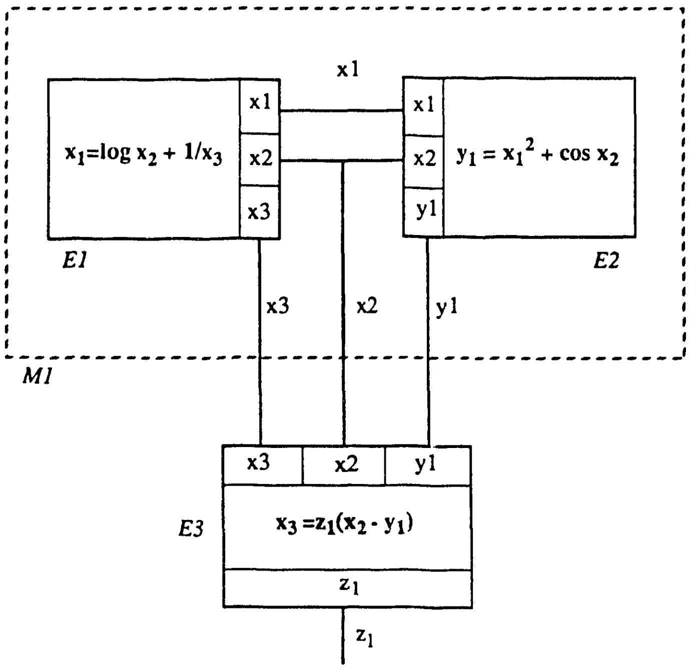


Figure 3: Linking of elementary SPARK objects to represent a system of equations. In this example, elementary objects *E1* and *E2* are linked to form a macro object, *M1*, which is then linked to *E3*, another elementary object. The links are the variables that are shared among the equations.

Due to the equivalence between equations and objects in SPARK, a system of equations can be described as elementary objects hooked together, as shown in Fig. 3. Describing such an object requires creating all of the elementary objects and their associated C functions and linking the elementary objects into a "macro" object.

To simplify this process, we have developed a MACSYMA-based symbolic interface to SPARK [Sowell 1990]. With this interface, you need only type in the equations for the system that you want to model. The interface consists of a set of commands that invoke MACSYMA functions to create the appropriate SPARK files. The arguments of these commands are equations in symbolic form, names (character strings), lists of names, etc.

In the following, we describe how you use the interface to easily create an elementary object (which corresponds to a single equation), a macro object (which corresponds to a system of equations), a dynamic object (which invokes an integrator), a dynamic macro object (which represents a system of ordinary differential equations), and a complete simulation (which includes objects, associated functions, simulation file, and input file).

3.1 Generation of elementary objects

The simplest object handled by SPARK is a single algebraic or transcendental equation, with no time derivative, but which can be piecewise defined on the variables' space.

The following command creates an elementary object:

makespark (eq, "name", badlist);

where *eq* is the equation in symbolic form (with a bracketing syntax in case it is piecewise defined), and *name* is a string that names the created object. The last argument, *badlist*, is a list of bad inverses, i.e., a list of variables that the user does not want the equation to be solved for. This list can be quite useful in speeding up the generation of object functions and in taking into account previous knowledge that some variables are bad iteration variables.

As an example of *makespark*, consider the equation for infrared radiation exchange between two surfaces of temperature T and T_0:

$$q_{12}=\epsilon(\theta,\phi)(T^4-T_0^4)$$

where $\epsilon(\theta,\phi)$ is the emissivity of the surface as a function of the direction that the radiation leaves the surface. The following command will make this into a SPARK object called *my_rad.obj* (see Fig. 4), treating the variables θ and ϕ as input parameters, but retaining the ability to calculate the temperatures or the flux:

makespark (q12=eps(th,phi)(t^4-t0^4), "my_rad", [th,phi]);*

Here, *eps* is an external function invoked by the generated C code. It is assumed to be present in the function library or embedded by the user as an internal function in the generated C code.

As an additional example of *makespark* — for the case of a piecewise defined equation — we consider the above equation for radiation exchange with a linear simplification for small temperature differences:

$$q_{12}=\epsilon(\theta,\phi)(T^4-T_0^4) \text{ if } |T-T_0|>\Delta T$$
$$q_{12}=h_{eq}(\theta,\phi)(T-T_0) \text{ if } |T-T_0|\leq\Delta T$$

where $h_{eq}(\theta,\phi)$ is the equivalent heat transfer coefficient in case of radiative linearization. The command for the creating the new object is:

```
makespark ([[q12=eps(th,phi)*(t^4-t0^4),(T-T0)^2>delta^2],
          [q12=h_eq(th,phi)*(t-t0),(T-T0)^2<=delta^2],"my_rad2",[th,phi,delta]);
```

Here, *h_eq* and *eps* are external C functions present in the function library. Note that *delta* is a bad inverse since it would an input to any problem using this equation.

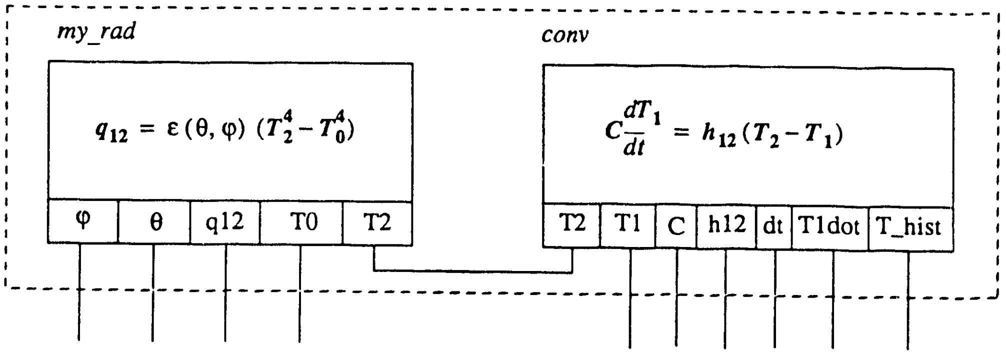

my_rad
conv
$q_{12} = \varepsilon(\theta, \varphi)(T_2^4 - T_0^4)$
$C\frac{dT_1}{dt} = h_{12}(T_2 - T_1)$
φ
θ
q12
T0
T2
T2
T1
C
h12
dt
T1dot
T_hist

Figure 4: SPARK macro object representing coupled radiative and convective heat transfer. *Conv* is a dynamic object, which, in SPARK, corresponds to a single ordinary differential equation.

It is worth noting that the notion of "bad inverse" is taken into account in modern generic engineering component description languages such as the Neutral Model Format [Sowell 1989]. It is based on the knowledge that a variable to which the overall system behavior is almost insensitive will be a bad variable to iterate on. Of course, the choosability of a bad variable is made possible by the fact that the SPARK environment does not force any variable to be input or output until specifically told so by the user. Therefore the objects available in the library after generation are essentially undirected. Specifying bad inverses is a way to limit this lack of directionality. Maximum limitation would occur by making all the equation variables bad except for one, which variable would then become the privileged output of the object. This would reduce the SPARK object to a classical subroutine with several inputs and one output.

The C function code that is generated when the elementary SPARK objects are created has several noteworthy features:

Along with the equation resolution code of the piecewise defined equation, satisfaction of the constraints is verified. This is implemented by adding to the domain condition the requirement that the obtained solution is within the domain.

When the domain supplied by the user does not entirely cover the variable domain, the remainder domain is handled by supplying as a default the solution found on the last portion of the variables' domain that was treated.

In case no solution is found, a default value is returned along with a warning message. This is sometimes useful to prevent the global Newton-Raphson iteration from failing in a region from which it cannot recover. Returning default values is acceptable as long as that happens only during the intermediate iterations and does not take place at final iteration time, when the final result is actually computed.

There is an additional safeguard option, that automatically generates code within the body of the generated C function, ensuring that no division by 0 occurs. In practice, this amounts to solving for all denominators in the expression and determining the conditions on the variables for which the denominators are zero. Then code is generated that flags these conditions and returns default values should they occur. The main purpose of these checks is to avoid a simulation that would be lost in numerical exceptions without chance of recovery. They amount essentially to resetting a break variable during iteration.

We have also implemented related safeguards, such as checking that the arguments of special functions (like log, asin, tan) are within range. An out-of-range argument is localized at run time precisely in the object in which it occurs.

We are considering extending these safeguards to automatic returning of limits or flagging of discontinuities in the functions. The first would be helpful when mathematical singularities correspond to no physical singularity. The second would be a useful warning for solutions techniques that rely on the continuity of functions or their derivatives (for example the Newton-Raphson method).

3.2 Generation of macro objects

A *macro object* is a system of equations, each of which may be piecewise defined. Macro objects are useful for representing complex physical components. In building science, for example, a typical HVAC component, such as a heat exchanger, will have several conservation laws to satisfy plus some constitutive behavioral equations. The number of equations depends on how detailed the model is. Describing an entity as a macro object allows the user to treat the entity as a whole — to instantiate it and link it to other objects — without having to worry about its internal details.

The following command creates a SPARK macro object along with its subobjects (the elementary objects corresponding to the individual equations) and associated C-language functions:

writemacro (sys, name, listofbadlist);

Here, *sys* is the equation system in symbolic form and *name* is either a string that names the macro object, or a list of names (if the user wants to choose the names of the subobjects), and *listofbadlist* is a list of bad inverses for each equation in the system.

Alternatively, to reuse objects that already exist in the SPARK library, *name* can be a list of the form

[name, [name1,[x=y,x2=y2]], [name2,[x1=z3,t=y,..], ...]

The symbolic interface will then generate the macro object by instantiating existing objects instead of creating new objects. In this case, the first argument of the list is the name of the macro object to be generated, and *name1*, *name2*, etc., are the names of the existing objects.

The list of equalities after each *namei* is a substitution that maps the names of the global variables in the macro object to the names of the local variables in the existing objects. In the example above, existing object *name1* has local variables called y and *y2* and the corresponding variables in the macro object are called x and *x2*. Thus, the macro object that is generated will contain statements of the form:

link x(name1_inst.y)
link x2(name1_inst.y2)

where *name1_inst* is the name of the instantiation of object *name1.*

The effect of *writemacro* is to put an equation system in network form. It scans the equations for common variables, states the links between equations with common variables, and generates all the elementary objects associated with the individual equations, along with the C functions that solve the equations for particular variables.

For example, consider the equation system

$$q_{12}=\epsilon(\theta,\phi)(T^4-T_0^4)$$
$$q_{12}+q=0$$

where the first equation is not already in the object library, but the second equation is (under the name *minus.obj*, with equation $in+out=0$). Then the following command will generate a macro object named *big_rad.obj* that corresponds to this system, plus new elementary object *my_rad.obj* and old elementary object *minus.obj*:

writemacro ([[[q12=eps(th,phi)(t^4-t0^4)]],[[q12+q=0]]],*
["big_rad","my_rad",["minus",[q12=in,q=out]]], [[th,phi],[]]);

Here we have specified that θ and ϕ are bad inverses in *my_rad*; we have specified no bad inverses for *minus.*

3.3 Generation of elementary dynamic objects

An elementary dynamic object corresponds to a single ordinary differential equation (ODE), possibly piecewise defined. The corresponding SPARK representation actually consists of two (or more) equations: the ODE itself (with the derivative given a variable name, for example *xdot*) and the integrator equations, which state that *xdot* is the derivative of x, *ydot* is the derivative of y, and so on. Thus the SPARK object will actually be a macro object with two (or more) subobjects and associated C functions. The command for creating an elementary dynamic object is:

makedynspark (eq, name, badlist, dynlist)

where the first three arguments are the same as those in *makespark* and the last argument, *dynlist*, is a list of pairs *[[x,xdot],[y,ydot],...]* indicating that *xdot* is the derivative of x, etc.

Typically, a dynamic object would be a first-order nonlinear differential equation. For example, the following command will create a dynamic object, *conv.obj* (see Fig. 4), for the heat transfer equation

$$C\frac{dT_1}{dt}=h_{12}(t_2-t_1)$$

for the case that the heat capacity, C, is always an input parameter (i.e., is never calculated):

*makedynspark (c*T1dot=h12*(T2-T1), "conv", [c], [T1,T1dot])*

Here *T1dot* is the time derivative of *T1*.

The time integrator used in this implementation is the backward difference integrator of order 4. It is an object. Thus *makedynspark* is nothing more than a *writemacro* involving an existing integrator object. Thus, it is very easy to change the integrator, provided its interfaces remain the same. For more complex integrators, it might be advisable to write the object by hand. However, code has been written that allows automatic implementation of the Runge-Kutta method of order 4 on a dynamic object. The command is:

makerungespark (eq, name, var, vardot)

or

makerungespark (eq, name, [var,vardot], dynvarlist)

where *dynvarlist* is the list of dynamic variables in the equation.

3.4 Generation of dynamic macro objects

A dynamic macro object represents a system of differential-algebraic equations. The command for creating this kind of object is:

writedynmacro (sys, name, listofbadlist, dynlist);

where the first three arguments are the same as for *writemacro*, and the fourth argument is a list of lists of the same type as *dynlist* in *makedynspark*.

Most components encountered in building thermal modeling are of this kind; an example is transient heat conduction through a wall discretized into several nodes.

3.5 Generating a simulation

If the user does not intend to link any objects together himself in the overall simulation file, and wants everything to be created for him, then the following syntax can be used:

writesimul (eq, name);

where *eq* is the overall equation system, including the equations that give the values of the inputs, and *name* is the name assigned to the overall simulation file. The *writesimul* command creates everything that is needed for a SPARK simulation to be ready to run, including the simulation and input files. This utility allows SPARK to be used simply as an equation solver, with no attention given to the crafting of objects or their reusability. This approach is convenient but inefficient, since the overhead for code generation is not exploited by reusing the code.

3.6 Component merging utility

It is sometimes numerically useful to eliminate variables from an equation system before making it into a SPARK object. That is especially true in the simulation of nonlinear controls, where numerical difficulties can occur.

Also, some internal variables of a macro object may of no interest, and are just calculation intermediaries. To eliminate such variables, a graph theoretic method applied to symbolic equations is used [Nataf 1987]. The command is:

reducer (sys, varlist);

where *sys* is the equation system and *varlist* the list of variables to be eliminated. The goal is the same as that of the MACSYMA command *eliminate*, but *reducer* is more versatile: MACSYMA's *eliminate* uses resultants, and is therefore suitable only for polynomial systems, whereas *reducer* can handle any type of equation.

In *reducer*, the ability of the equations to be solved in terms of their variables with inverse functions is used in order to perform heuristically chosen substitutions. The algorithm transforms the equation system into a graph in which the equations are arcs and the variables are nodes. Two nodes are connected by an arc if a variable is shared by two equations. In general, there are several arcs associated with each node, since most equations will have more than two variables.

At this point, the variables that are present in many equations are saved for later substitution, since substituting them might make implicit many equations where they are used. The reason for this is that a pessimistic heuristic approach is taken, and it is assumed that substitutions into equations make these equations more complicated, and probably implicit in the variables that were injected. Although this is not always the case, it provides a rough guideline on how to choose variables to substitute.

The criteria for choosing the order of the variables to substitute and the equations to use for the substitution are described in more detail in [Nataf 1987]. These criteria have been implemented in both FORTRAN and MACSYMA. The MACSYMA implementation is slower but more efficient since the substitutions are actually performed and advantage is taken of the resulting simplifications, whereas the FORTRAN implementation can only take the pessimistic hypothesis that substituting a variable in an equation will make it implicit in the new variables that are injected. For example, substituting $y=x^5$ into $z=x+y+1$ (explicit in x) yields $z=x^5+x+1$ (implicit in x).

3.7 Generation of macro object networks

Some equation systems have a particularly simple and repetitive form. In heat transfer, for example, the electrical analogy for conductive, convective and radiative transfer leads to equation systems of a simple form:

$$c_k = \sum_{i=1}^{N} a_{ik} b_i$$

where b is any expression with index i (a vector) and a is any expression with two indices (a matrix). This approach can be used, for example, for conveniently expressing the equations for the radiative interaction between plane surfaces for which the shape factors have two indices and the flux has one index.

The SPARK command for generating macro object networks is:

writegenericnetmacro(n, name, objname, expr1, expr2, badinvlist);

where n is the number of equations, *name* is the macro name, and *objname* is the name of the elementary object describing the equations. These equations have the form

$$expr\,1 = \sum_{j=1}^{n} expr\,2$$

where *expr1* depends on index k and *expr2* depends on indices k and j.

The last argument, *badinvlist*, is the list of bad inverses for the equation associated with $k=1$. It is assumed that this list is also valid — by "symmetry" — for the other instantiations of the equation $expr\,1 = \sum_{j=1}^{n} expr\,2$. It must be remembered that only one elementary object is created for that equation, and that that object is then instantiated n times. But since bad inverses can only be excluded in elementary objects, the ist of bad inverses is supplied only for the equation associated with $k=1$.

There is a noteworthy problem associated with network objects: they are unphysical in the sense they can only be used when connected to other objects. For example, if we create a radiative exchange network object for a room with N walls, we will have to connect it to the wall surfaces between which the radiative exchange takes place. If we now add a new wall, then the original N-wall radiative network object is no longer valid, and has to be replaced with an (N+1)-wall radiative network object.

One could argue that this problem is due to the matrix representation method chosen, and that one does not need to represent the interaction as an object, but can put the radiative behavior in the walls themselves. But then each wall will have a radiative influx interface that will have to be the sum of the exchanges with the other walls, the number of which is not known *a priori*. So the problem remains. Therefore, when adding a wall, one has to change the global radiation object, or one has to change the wall objects. The first alternative shows that there can be no general radiation object in SPARK. The second alternative shows that there can be no general radiative wall object in SPARK.

This type of difficulty arises because a variable environment cannot be parametrized *a priori* and put into an interface. An interface is scalar, and the number of scalar variables through which a SPARK object communicates with the outside is fixed.

In practice, it is usually possible to deal with these problems, since the number of walls (continuing with the above example) always ends up being instantiated to a fixed value. Difficulties might arise, however, when new objects appear during a simulation and cause the number of equations to change. In principle, SPARK cannot handle this. But it may be possible to write the equation system so that at certain times certain equations, although present, are irrelevant, and only take on a physical meaning when the system reaches a certain state. In this case, the "extra" equations are fired during the iteration process even if the current state is inappropriate for them, but their effect does not influence other results. This approach has been proven to be feasible (for example, for a coil that switches from heating, with no condensation, to cooling, with condensation and therefore with additional equations) but requires extreme care in designing a system of equations that will, under certain conditions, behave as a smaller system, with the complementary subsystem not influencing the part that remains of physical significance.

3.8 Generation of a simulation containing two-dimensional PDE's

SPARK handles systems of algebraic and ordinary differential equations, but has no built-in way to treat partial differential equations (PDE's). Two approaches to handling PDE's in SPARK are illustrated in Fig. 5. One approach is to resort to approximate closed-form solutions of the PDE, as determined, for example, by a variational method. The resulting equation is then used to create a SPARK object using the symbolic interface, as described above.

A second approach is to observe that a finite difference representation of a problem yields a system of differential-algebraic equations that is well suited to treatment with the SPARK object-oriented methods. In the 2-D finite difference discretization, each elementary bulk domain rectangle can be described by the same object (see Fig. 6). Furthermore, there are only a few possible configurations for the boundary elements (primarily corners and flat boundaries), which means that only a few types of objects are needed to represent all possible boundary conditions (see Fig. 6).

The finite-difference approach has been implemented in SPARK. It handles second-order PDE's with first-order boundary conditions on 2-D domains of any shape that is regular enough. However, no provision is made for the error due to approximating a smooth boundary with rectangles, although a wider variety of boundary objects could be implemented automatically*.

For steady state, the command is:

writefindiff2Dsimul (name, objname, bcname, diffeq, domain, constraint, dx,dy, badlist);

where *name* is the name of the overall simulation file, *objname* is the bulk cell object name, *bcname* is the suffix for the boundary condition object name (prefixed with *x_*, *y_* or *xy_* depending on whether it is a left/right, top/down or corner boundary), *diffeq* is the PDE, *constraint* specifies the boundary conditions, and *domain* is a 2-D function that is negative inside of the domain and zero at the boundary. The quantities *dx* and *dy* are the spatial discretization steps in the *x* and *y* dimension, and *badlist* is the list of variables that we do not want to solve for (either because they will be parameters and we will always input them, or because solving for them is very time consuming for MACSYMA, or because they exhibit bad numerical properties as iteration variables).

The syntax for dynamic or dynamic vectorial PDE's is the same but a slightly different package is invoked.

SPARK can therefore handle problems as complex as natural convection in two dimensional enclosures, for example. You need only enter the PDE in symbolic form, together with the boundary conditions and domain geometry. The C code needed to simulate the problem is then automatically generated. However, only fairly coarse grids can be handled, otherwise the solver that is generated may be very slow to compile or may exceed the capabilities of the compiler.

The user has the option to specify alternative algorithms for discretization. In the code the discretization is described by rules. The user can change these rules, and not further modify the code, provided that the value at each point in the domain is influenced only by its immediate neighbors.

An example of using *writefindiff2Dsimul* for generating the simulation of 2-D heat conduction in a disk in shown in Section 5.

*This is to take into account the fact that the rectangular cells overlap the real domain. One reason is simplicity: the smaller the number of boundary objects the better (here we need only side and corner objects), even though one might have to resort to a finer grid to reduce the discretization errors. Another reason is that body-fitted coordinates can be used to transform the domain into a collection of rectangular domains, as described in [Thompson 1985], thus eliminating entirely the need to deal with the irregular boundary case in the computational cells themselves.

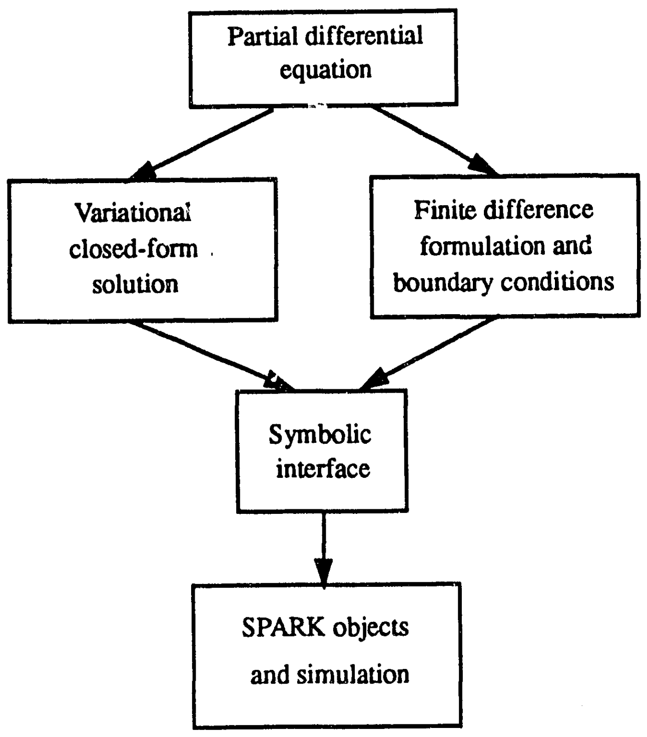

Partial differential equation
Variational closed-form solution
Finite difference formulation and boundary conditions
Symbolic interface
SPARK objects and simulation

Figure 5: Two approaches for handling partial differential equations in SPARK.

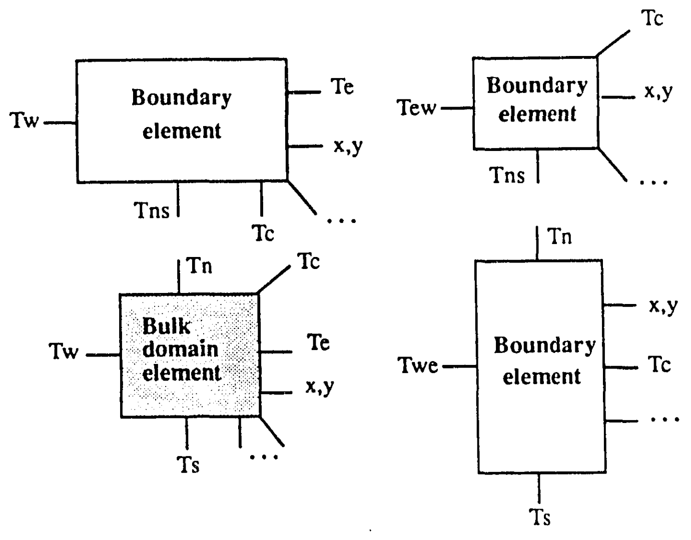


Figure 6: Finite difference objects in SPARK.

4. Use of Compiler-Compilers

In this section we describe the use of compiler-compilers for code generation in SPARK.

4.1 Multilinks

Originally, the links connecting SPARK objects were restricted to be scalar quantities. To link objects with vector quantities required specifying separate links for each component of the vector. For example, the exchange of a fluid characterized by a (temperature, mass-flow-rate) vector required separate temperature and mass-flow-rate links. To get around this problem, the "multilink" concept was introduced.

A multilink is an array of scalars (or other multilinks, to any depth). Multilinks were implemented by extending the SPARK Network Simulation Language syntax and writing a lexical analyzer and syntactical parser, using Lex [Lesk 1986] and Yacc [Schreiner 1985 and Johnson 1978], to translate the new syntax into the old, SPARK-compatible syntax. The Lex and Yacc utilities take as input formalized descriptions of atoms and grammar, and generate C programs that do the parsing of any file for these language specifications. The SPARK language, being very simple, is suitable for treatment by these utilities, which can be used for writing compilers or translators (the latter being the case here). The Lex and Yacc generated parser just has to be modified slightly, since it is not recursive, while the SPARK parser is (since it has to deal with embedded macros). This treatment (which involves adding and maintaining a "depth of recursion" dimension to the internal arrays of the generated parser) is all automated.

In the extended syntax, an *mport* statement defines a multilink. For example, the statement

mport air(h, db, w)

will have the parser understand that linking two air-flow interfaces called *air* together means actually linking three separate interfaces — specific enthalpy *h*, dry bulb temperature *db*, and humidity ratio *w*. Thus, a statement of the form

link air45 (tube1.air_out, coil2.air_in)

will be expanded into:

link air45->h (tube1.air_out->h, coil2.air_in->h)
link air45->db (tube1.air_out->db, coil2.air_in->db)
link air45->w (tube1.air_out->w, coil2.air_in->w)

The nesting feature allows the elements of an *mport* to be other *mports*, to any depth.

4.2 Parametrized macro objects

Another limitation of the SPARK approach is that each object has a fixed number of interfaces. This is bothersome when dealing with "parametrized objects," an example of which is a wall with *N* layers, where *N* is not known *a priori*.

To overcome this limitation a preprocessor was built, using Lex and Yacc, that allows the user to create parametrized objects and simulations using an indexed syntax, with *N* as a variable. Upon instantiation of that variable, the preprocessor creates a C program that asks the user to specify the value of *N*, and then generates a MACSYMA program that in turn generates the associated SPARK simulation or macro object file.

4.3 Steady-state resetting of a dynamic simulation

A dynamic simulation may fail to converge at a particular time step. There are various possible reasons for this, including bad problem conditioning, too large a time step, or sudden change of a parameter. To deal with this, a method has been devised for SPARK using Lex and Yacc that — upon non-convergence — automatically creates a "ghost" steady-state simulation by removing all time derivatives from the dynamic simulation and assigning input parameters to be those at the time of failure. The dynamic simulation is then restarted at this time step using as new starting values the results of solving the steady-state problem. This approach can often save simulations that can otherwise be made to converge only by resorting to unreasonably small time steps.

5. Example of the Symbolic Interface: 2-D Conduction

We consider the SPARK simulation of heat conduction in a disk. We take the disk to be a section through an infinitely long rod, so that the heat conduction is in the two dimensions perpendicular to the axis of the rod. The disk has a heat production term in the bulk domain and a Newtonian convection boundary condition. The conduction equation to be solved is

$$\Delta T + \frac{u}{k} = \rho c_p \frac{\partial T}{\partial t}$$

where T is the temperature,u is the bulk heat generation rate, k is the thermal conductivity, ρ is the density, and c_p is the specific heat capacity.

The boundary condition on the perimeter of the disk is

$$-k\frac{dT}{dn} = h(T - T_0)$$

where h is the heat transfer coefficient, T_0 is the ambient temperature, and $\frac{dT}{dn}$ is the normal derivative of the temperature at the boundary.

The SPARK commands for generating the simulation for this problem are as follows:

```
/*p2dyn simulation*/
/*Disk with uniform heat generation, Newtonian convection loss*/
batch("/u1/nataf/vaxima/mysolve.mac");
batch("/u1/nataf/vaxima/FINDIFF2D/findiff2Ddyn.mac");
r0:1.0;
circle1(x,y):=x^2+y^2-r0^2;
ctt:[[-k_avg/r0*('diff('temp,'x)*'x('x,'y)+'diff('temp,'y)*'y('x,'y))=h_avg*('temp('x,'y)-temp0)]];
eqdif:[[k_avg*('diff('temp,'x,2)+'diff('temp,'y,2))+'u_avg=rho*cp*'diff('temp,'t)]];
writefindiff2Dsimul ("p2dyn", "p2dynelt", "bcp2dynelt", eqdif, circle1,
   ctt, 0.1, 0.1, [temp0,rho,cp,k_avg,u_avg]);
closefile();
```

The SPARK solution for the disk temperature distribution at 0, 1, 2, and 10 sec is shown in Fig. 7 for the following parameters: disk radius (r_0) = 0.1 m, k=0.032 W/(mK), u=10000 W/m^3, h=400 W/(m^{2}K), ρ=1020 kg/m^3, c_p =0.24 J/(kg-K), initial disk temperature = 24 C, and ambient temperature = 20 C.

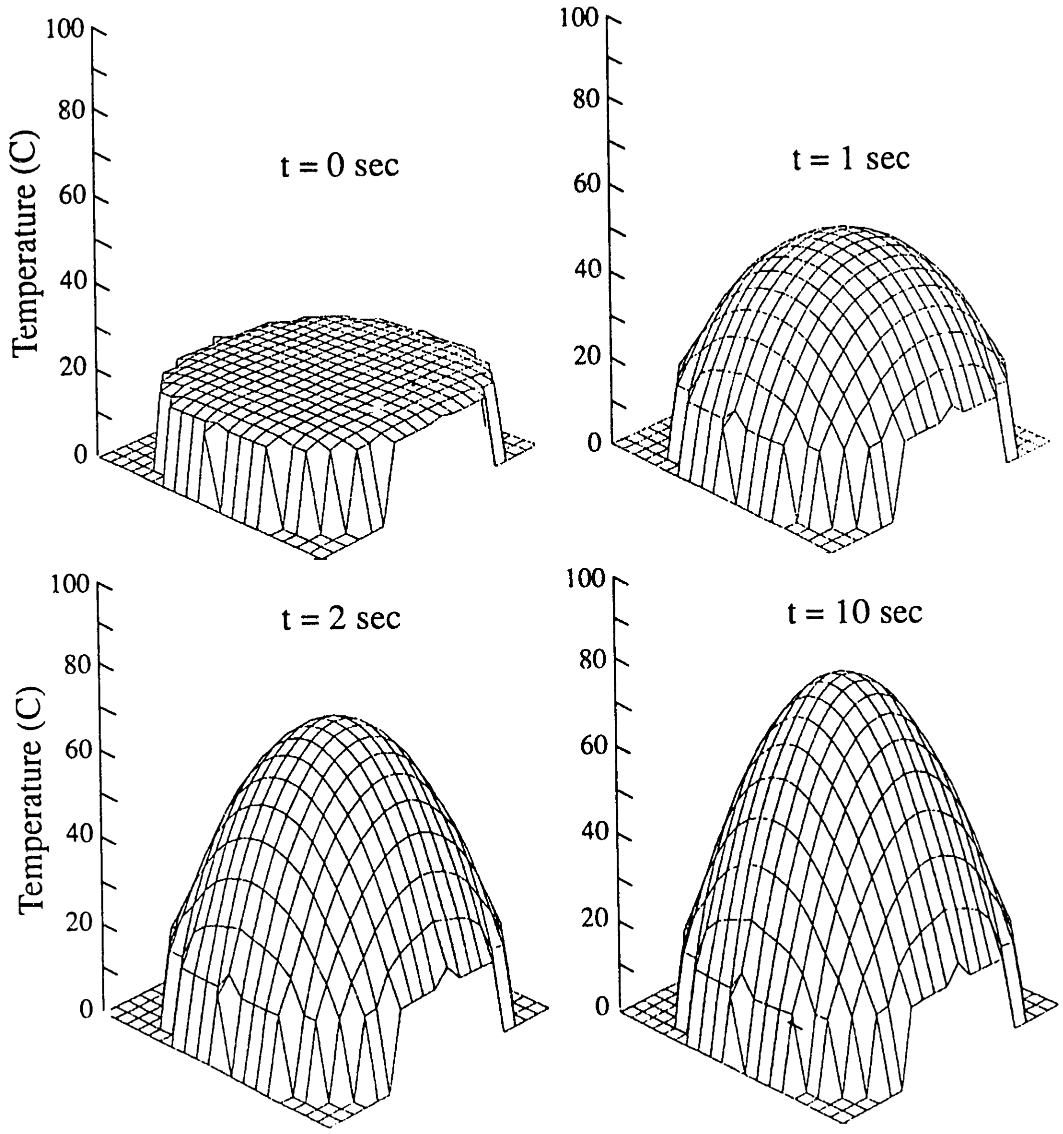


Figure 7: SPARK solution for disk temperature distribution.

For this problem, the reduction factor is 1 (i.e., no reduction, which means there are as many iteration variables as there are dynamic variables) or infinity (which means there are no iteration variables), depending on whether all of the dynamic variables are in the cut set by construction, as in an initial implementation of SPARK, or not, as in a new version where this constraint has been eliminated.

The MACSYMA input needed is fairly short, but generates a lot of reusable SPARK and C code. The simulation that is automatically generated has 1741 objects or links, and 285 break variables. Thus, the number of iteration variables is quite large. An alternative implementation, which does not force the dynamic variables to be break variables, leads to zero iteration variables! The reason for this is that the integrator used is explicit. Hence, initial conditions are enough to explicitly calculate all unknowns at each time step using only the unknown values at the previous time step. However, in this case the simulation is sensitive to the usual stability criteria between time step and grid size, while the previous code is not and converges to the right values even outside the usual stability domain. This is to be expected: iteration on all dynamic variables leads to a resolution process immune to the stability problems occasioned by forward time and center space differencing.

Conclusion

The SPARK environment provides a convenient basis for quick prototyping of simulation programs. Its object-oriented interface makes it suitable for component-based simulations of the kind encountered in heat transfer and thermal engineering. SPARK's symbolic preprocessors reduce model-building time, generate component libraries automatically, and permit automatic generation of solutions to complex PDE problems. The implementation of these problems is not necessarily the most efficient, since the SPARK idea of efficiency is only based on equation system size reduction. It is not necessarily the most accurate either, since a general purpose discretization is used that does not consider the physical characteristics of the problem.

Introducing physical insight in the choice of numerical schemes by integrating an expert system into SPARK is under consideration. Under development are symbolic graph theoretical tools for reducing equation subsystems and for generating customized simulations (discretizing PDE's with arbitrary boundary conditions, for example).

References

Anderson, J. L. 1986. *A Network Definition and Solution of Simulation Problems*, Lawrence Berkeley Laboratory report no. LBL-21522.

Buhl, W.F. et al. 1990. *The U.S. EKS: Advances in the SPANK-based Energy Kernel System*, Proc. Third International Conference on System Simulation in Buildings, Liege, Belgium.

Johnson, S.C. 1978. *YACC: Yet Another Compiler Compiler*, in UNIX Programmer's Manual, vol. 2, Bell Telephone Laboratories, Inc., Murray Hill, N.J.

Lesk, M.E. and E. Schmidt 1986. *LEX, A Lexical Analyzer Generator*, in UNIX Programmer's Manual, vol. 2, Bell Telephone Laboratories, Inc., Murray Hill, N.J.

MIT 1983. *MACSYMA Reference Manual, version 10,* Mathlab Group, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA.

Nataf, J.-M. 1987. *Automatic Modeling of Thermal Systems*, Dissertation, Mechanical Engineering Department, Univ. of California, Berkeley.

Nataf, J.-M. and F. Winkelmann 1991. *Dynamic Simulation of a Liquid Desiccant Cooling System using the Energy Kernel System*, Lawrence Berkeley Laboratory report no. LBL-29610.

Sowell, E.F., and P. Sahlin 1989. *Neutral Format and Automatic Translation for Building Simulation Submodels*, Proc. Building Simulation '89, Vancouver.

Sowell, E.F. and J.-M. Nataf 1990. *Radiant Transfer due to Lighting: an Example of Symbolic Model Generation for SPANK*, Proc. Society for Computer Simulation Western Multiconference, San Diego, and Lawrence Berkeley Laboratory report no. LBL-28273.

Schreiner, A.T. and G.H. Friedman, Jr. 1985. *Introduction to Compiler Construction with UNIX*, Prentice-Hall, Inc., Englewood Cliffs, N.J.

Thompson, J.F. et al. 1985. *Numerical Grid Generation: Foundations and Applications*, North-Holland, Elsevier Science Publishing Co., New York.