SPARK *2.0*
REFERENCE MANUAL

Simulation Problem Analysis and Research Kernel

This work was supported by the Assistant Secretary for Energy Efficiency and Renewable Energy, Office of Building Technologies Program of the U.S. Dept. of Energy. Contract No. DE-AC03-76SF00098.

TABLE OF CONTENTS

FOREWORD

Documentation for the *SPARK* program is comprised of two manuals: the *SPARK* Reference Manual and the *VisualSPARK* Users Guide. These documents are available as downloadable PDF files from http://SimulationResearch.lbl.gov.

This Manual is intended to cover the basic principles of *SPARK* programming. To the extent possible, it is intended to be independent of the platform. Consequently, examples are demonstrated using the command line interface only.

This work was supported by the Assistant Secretary for Energy Efficiency and Renewable Energy, Office of Building Technology, State and Community Programs, Office of Building Systems of the U.S. Department of Energy, under contract DE-AC03-76SF00098.

The *SPARK* simulation program is not sponsored by or affiliated with SPARC International, Inc. and is not based on SPARC architecture.

LICENSES AND COPYRIGHTS

UMFPACK Version 4.0 (Apr 11, 2002). Copyright (c) 2002 by Timothy A. Davis. All Rights Reserved. *UMFPACK* License: Your use or distribution of *UMFPACK* or any modified version of *UMFPACK* implies that you agree to this License. THIS MATERIAL IS PROVIDED AS IS, WITH ABSOLUTELY NO WARRANTY EXPRESSED OR IMPLIED. ANY USE IS AT YOUR OWN RISK. Permission is hereby granted to use or copy this program, provided that the Copyright, this License, and the Availability of the original version is retained on all copies. User documentation of any code that uses *UMFPACK* or any modified version of *UMFPACK* code must cite the Copyright, this License, the Availability note, and "Used by permission." Permission to modify the code and to distribute modified code is granted, provided the Copyright, this License, and the Availability note are retained, and a notice that the code was modified is included. This software was developed with support from the National Science Foundation, and is provided to you free of charge. Availability: http://www.cise.ufl.edu/research/sparse/umfpack

Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd and Clark Cooper. Copyright (c) 2001, 2002 Expat maintainers. Availability: http://www.libexpat.org/

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the"Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions: The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

TEXT CONVENTIONS

Throughout this manual, we use different typefaces as follows:

Program Name

File Name

KEYWORD

```
Screen Display, Code, Key
```

In addition, when discussing *SPARK* terminology (starting with Section 2.1), italic and bold typefaces identify the different entities, as follows:

problem name	**macro class**
object name	**atomic class**
probe, link name	**port name**
problem variable	**port variable**

1 INTRODUCTION

1.1 WHAT IS *SPARK*?

Simulation of a physical system requires development of a mathematical model that is usually composed of differential and/or algebraic equations. These equations must then be solved at each point in time over some interval of interest. The Simulation Problem Analysis and Research Kernel (*SPARK*) is an object-oriented software system that performs such simulations. By object-oriented we mean that components and subsystems are modeled as objects that can be interconnected to specify the model of the entire system. Often the same component and subsystem models can be used in many different system models, saving the work of redevelopment.

1.2 KINDS OF PROBLEMS THAT *SPARK* CAN SOLVE

Since nearly any physical or biological system can be described in terms of a mathematical model, SPARK can be used in many scientific and engineering fields.

SPARK may be thought of as a general differential/algebraic equation solver. This means that it can be used to solve any kind of mathematical problem described in terms of a set of differential and algebraic equations. The term "continuous system" is often used to describe this class of problems. Typical examples include building heating and cooling systems, heat transfer analysis, and biological processes.

While, in principle, any system can be described in terms of differential and algebraic equations, there are many systems that are more easily described in terms of discrete states. Typical examples include assembly lines from the field of manufacturing engineering and queuing problems from various fields. *SPARK* is not designed for discrete state simulation problems. However, there are limited facilities for handling discrete events in otherwise continuous systems.

1.3 DESCRIBING PROBLEMS FOR *SPARK* SOLUTION

Describing a problem for *SPARK* solution begins by breaking it down in an object-oriented way (Nierstrasz 1989). This means thinking about the problem in terms of its components, where each component is represented by a *SPARK* object. A model is then developed for each component not already present in a *SPARK* library. Since there may be several components of the same kind, *SPARK* object models, i.e., equations or groups of equations, are defined in a generic manner, called classes. Classes serve as templates for creating any number of like objects that may be needed in a problem. The problem model is then completed by linking objects together, thus indicating how they interact, and specifying data values that specialize the model to represent the actual problem to be solved. Section 2 has several examples

Naturally, model descriptions must be expressed in some formal way. *SPARK* class models are described in a textual language that is similar to other simulation programming languages except that it is non-procedural. That is, it is neither necessary to put the equations in order, nor to express them as assignment statements. This property derives from the input/output free manner in which the classes are defined, and the use of mathematical graphs (McHugh 1990) to find an appropriate solution sequence.

In *SPARK*, the smallest programming element is a class consisting of an individual equation, called an atomic class and stored in a file with extension *.cc. Macro classes bring together several atomic classes (and possibly other macro classes) into a higher level unit. Macro classes are stored in files with the extension

*.cm. Problem models are similarly described, using a combination of atomic and macro classes, and placed in a problem specification file with extension *.pr.

Figure 1-1 shows the steps involved in the *SPARK* build process whereby the problem description expressed in the *SPARK* language is transformed into an executable program that can be executed to solve the problem for given boundary conditions.

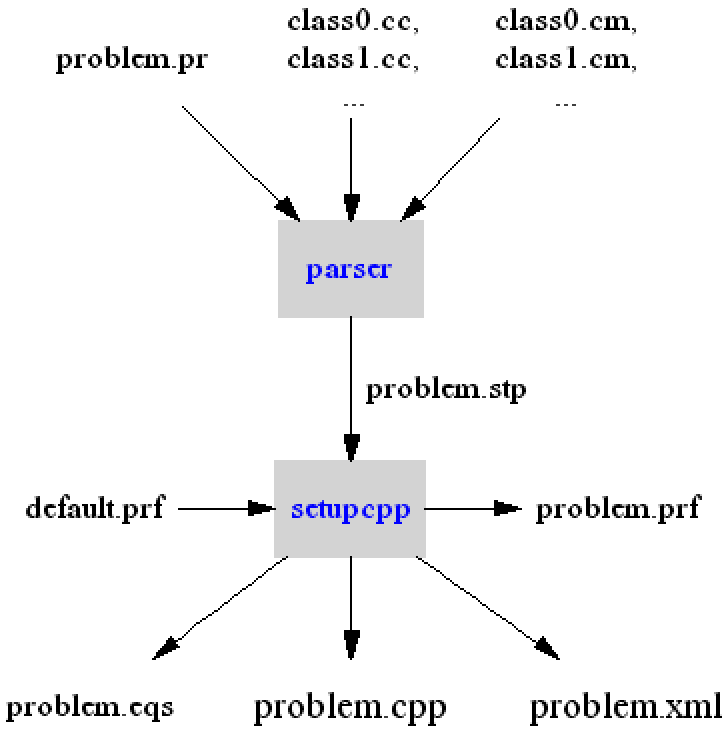

Figure 1-1: *SPARK* Build Process.

When the problem is processed by *SPARK*, the problem specification problem.pr file is first parsed along with the macro and atomic classes used in the problem, in order to generate the setup file problem.stp. The setup file contains a flat representation of the problem, which is then used for the graph-theoretical analysis performed by the *setupcpp* program. This step produces an efficient solution sequence for the underlying system of equations. The *setupcpp* program writes this information out in various files with different formats: the user-readable equation file problem.eqs, the *C++* file problem.cpp and the XML file problem.xml.

To produce an executable simulator the problem.cpp file can be compiled and linked against the solver's library. Another approach to generating an executable simulator consists in loading the problem description provided in XML format in the problem.xml file and instantiating the corresponding solver at runtime. The process of building the executable simulator is typically automated using a makefile or a build program in the *SPARK* installation. Finally, at runtime the preference file, problem.prf can be used to specify the settings for the various solution methods.

You must have access to a *C++* compiler on the machine running *SPARK*. On Windows 95/98/NT platforms, the default *WinSPARK* installation assumes that you have Microsoft *Visual C++* installed, but *Borland*, *GNU*, and *Symantec* compilers are also supported. *VisualSPARK* on Windows 95/98/2000/ME/NT platforms

usually uses the *mingw* implementation of the *GNU C++* compiler. UNIX installations normally use the *GNU* compiler, but *SPARK* has also been used with other compilers commonly available on Sun workstations.

While specifying problems in the *SPARK* language using existing classes is relatively easy, writing *SPARK* class models can be tedious. One necessary task is deriving the inverses for the class equation, i.e., closed-form solutions for several or all variables that occur in the equation. The labor of this task is multiplied in certain kinds of problems, such as those described in terms of partial differential equations. Such equations must first be expressed as sets of ordinary differential equations replicated many times with slight variations. To simplify this, *SPARK* can be installed with symbolic tools, such as *Maple* (Char, Geddes et al. 1985). With such tools you need specify only the atomic class equation, from which all necessary inverses and supporting *C++* functions are generated automatically through symbolic manipulation. For users without *Maple*, *SPARK* comes with its own symbolic manipulation tool that, while very limited, can find inverses of many equations encountered in simulation practice. For more involved problems, these symbolic tools offer a significant improvement in productivity. However, initially it will be more instructive for you to use *SPARK* directly, as we show here.

1.4 PORTABILITY AND USER INTERFACES

SPARK is intended to be portable. The basic elements, i.e., the parser, setup program, and fixed elements of the solver, will compile and run on nearly any platform for which there is a *C++* compiler. In the current release, executables, necessary source code, and graphical user interfaces are provided for the UNIX and Windows platforms. On both platforms, the graphical user interfaces allow text-based creation of classes and problems using the *SPARK* language, as well as problem execution and results display. Post processing for visualization of results is supported in both environments.

1.5 THE HISTORY OF *SPARK*

Although a general tool, *SPARK* was developed for use in the simulation of building service systems, e.g., heating, ventilation and air-conditioning. Most usage up to the time of this writing has been on systems from this field.

The first implementation of *SPARK*, which solved only algebraic problems, was done at the Lawrence Berkeley National Laboratory in 1986 (Anderson 1986). The basic ideas, including the graph-theoretic aspects, were based on earlier work at the IBM Los Angeles Scientific Center (Sowell, Taghavi et al. 1984). Then, in 1988, the LBNL implementation was extended to allow solution of differential equations (Sowell and Buhl 1988). The *MACSYMA* and *Maple* interfaces were developed by Nataf (Nataf and Winkelmann 1994), who also made many other improvements. Since that time there have been new developments. For example, the solver was revised to decompose the problem into separately solvable components (Buhl, Erdem et al. 1993).

Then in preparation for the initial public release (version 1.0), *SPARK* was completely rewritten in 1995-96. In this rewrite a new class and problem description language was implemented to improve modeling flexibility, and the solver was redesigned to improve solution speed. In addition, several user interface tools were developed, including a simple symbolic manipulation tool.

The current release (version 2.0) significantly extends the modeling capabilities of earlier versions by supporting multi-valued inverses which calculate multiple values simultaneously instead of a single value. A multi-valued inverse essentially models a multi-dimensional vector function. Also, it is now possible to attach private data to each inverse instance in the problem under study with the help of the added callback mechanism. Finally, real-time operation of a *SPARK* simulator is made possible by the addition of the request mechanism that allows to synchronize the solver's global time with user-specified meeting points at runtime. Furthermore, this capability along with the possibility of customizing the driver function facilitates integrating a *SPARK* simulator as a black-box solver in another application.

1.6 VERSIONS OF *SPARK*

A document, entitled README.txt, is included in the release package of *VisualSPARK*. The file is located in the doc subdirectory and describes new features, changes and bug fixes from the previous to the current version.

2 BASIC METHODOLOGY

Although *SPARK* is intended for the analysis of complex physical systems represented as large systems of nonlinear equations, both algebraic and differential, an understanding of the basic methodology can best be obtained by working first with simple mathematical problems. We begin with the simplest possible problem, a single linear equation. This problem is then extended in steps to demonstrate more and more *SPARK* features. This will prepare us for dealing with more complex systems in later sections.

2.1 OVERVIEW AND TERMINOLOGY

We begin by defining some terminology. The basic entity in a *SPARK* model is the **object**, it consists of a single algebraic equation that calculates one value and its interface or **port** variables. Objects are created by reference to a **class**, which may be thought of as a template for the equation object. As an example, consider the simple equation for the sum of two real numbers:

$$a + b = c \tag{2.1}$$

The class, which we might call **sum,** would contain this equation (2.1), and its ports would consist of the variables *a, b,* and *c* . Figure 2-1 is a pictorial representation of this idea.

Note that we distinguish between an object and the class from which it was created. This is because there might be need for more than one equation of this form in a particular model. We can create as many instances (objects) from the class **sum** as we wish. Moreover, classes are saved, allowing their use in many different problems. In this way, *SPARK* reduces the model development work through code reuse.

Note also that the possibility of multiple instances of a class means that we must distinguish between the symbols used in defining the class and the corresponding variable names occurring in the problem definition. That is, if we wish to have the **sum** class represent both $x + y = z$ and $r + s = t$, it is obvious that *a* must represent *x* in one place and *r* in another. We call variables such as *x* and *r* **problem variables** because they relate to a particular problem being described. On the other hand, **a**, **b**, and **c** relate only to the class definition and are called interface or **port variables**. It is also common to refer to *SPARK* problem variables as **links** because the keyword `LINK` is used to connect object ports, thus introducing the variable and assigning to it a name. We will see this in examples that follow.

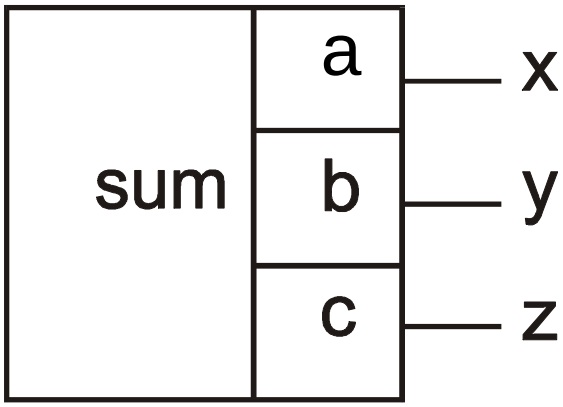

Figure 2-1: **sum** class diagram

In addition, when discussing *SPARK* terminology, italic and bold typefaces identify the different entities, as follows:

problem name	**macro class**
object name	**atomic class**
probe, link name	**port name**
problem variable	**port variable**

2.2 A PROBLEM WITH A SINGLE OBJECT

As a first exercise we will develop a *SPARK* solution for a simple math problem called *2sum*. In *2sum* we seek solutions for the equation:

$$x + y = z \tag{2.2}$$

As we saw in Section 2.1, there is a class in the *SPARK* foundation class library globalclass called **sum** that we can use to solve this problem. As shown in Figure 2-1, its port variables are **a**, **b**, and **c**, and it enforces the relationship of Equation (2.1). Obviously, by associating **a** with *x*, **b** with *y*, and **c** with *z* we can represent Equation (2.2) with an object of the **sum** class.

Equation (2.2) is a mathematical model involving three variables and one equation. To create a well-posed problem, we have to define two inputs. For this example, let's specify *x* and *y* as input, so that *z* is to be determined. The problem definition file 2sum.pr then has the following contents:

```
/*   Problem Definition File for Simple Math Problem 2sum.pr */
DECLARE sum  s;
LINK    x s.a  INPUT  REPORT;
LINK    y s.b  INPUT  REPORT;
LINK    z s.c         REPORT;
```

Inputs are the quantities known at the outset.

Links that are not inputs are variables to be solved for. These variables are also referred to as the output variables or the unknowns.

Here the DECLARE statement creates an object *s* as an instance of the class **sum**. The LINK statements associate the problem variables with object port variables. The links *x* and *y* are associated with the corresponding object port variables **s.a** and **s.b** respectively. Note that we employ the notation **name.variable** to refer to the port variable of object name. The keyword INPUT[1] in the LINK statements indicate that these problem variables are inputs, as opposed to being determined by the solution process. A LINK statement with the keyword INPUT is also referred to as an INPUT statement. The LINK statements without the INPUT keyword are variables to be solved for rather than inputs. The keyword REPORT in LINK statements means that the variable should be reported in the *SPARK* output.

This results in the following directed graph for the *sum* problem with the specified input/output designation.

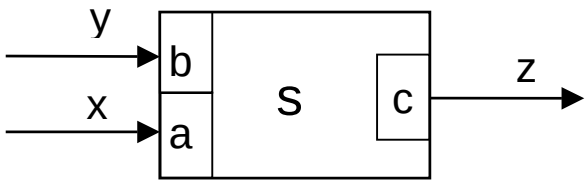

Figure 2-2: Directed graph for the *sum* problem.

After creating 2sum.pr as shown above, you must create an input file called 2sum.inp with the following contents:

[1] A LINK statement with the INPUT keyword can also be specified using the INPUT keyword in place of the LINK keyword. With this shorthand notation it is no longer necessary to repeat the INPUT keyword in the statement. E.g.:

```
INPUT   x s.a  REPORT;
```

```
2       x       Y
0       1       2
```

Here we see the format of a *SPARK* input file. The first line gives the number of input items, followed by their symbols as defined by the `INPUT` statements in the problem specification file. The subsequent lines give values for each input variable, preceded by the time at which these values apply. If the problem is not a dynamic one, i.e., we are seeking a solution for only one set of inputs, only two lines are required as shown above. However, if we seek solutions at other time values, as many lines as needed can be given. This is discussed further when we take up dynamic problems in Section 6.

2.2.1 Running the SPARK Problem

You can now run the problem with *SPARK*. The commands to do so differ somewhat depending on your installation and platform. For a *WinSPARK* installation on the Windows platform, type:

```
buildsolver 2sum.pr spark.prf <enter>
```

This creates an executable program called 2sum.exe Several other files are also created, including 2sum.prf and 2sum.run, that are needed to execute 2sum.exe. To execute the program for numerical solution enter:

```
sparksolver 2sum.prf 2sum.run 2sum.xml <enter>
```

If you are working with *VisualSPARK* on either the Windows or UNIX platforms, the equivalent command is:

```
runspark <enter>
```

This builds and executes the single allowed problem file in the current working directory. It can be executed again without rebuilding with the command line:

```
sparksolver 2sum.prf 2sum.run 2sum.xml <enter>
```

Since *SPARK* is often used to solve dynamic problems, run-control information is needed when the program begins to execute. This information is provided in a problem run-control file, probName.run, generated automatically when you first run a new *SPARK* problem. The file has the format of a *SPARK* preference file, discussed in Appendix C.

The run-control file for 2sum.pr, i.e., 2sum.run, is:

```
(
InitialTime          ( 0.0 ())
FinalTime            ( 0.0 ())
InitialTimeStep      ( 1.0 ())
FirstReport          ( 0.0 ())
ReportCycle          ( 1.0 ())
DiagnosticLevel      ( 3 ())
InputFiles           ( 2sum.inp ())
OutputFile           ( 2sum.out ())
FinalSnapshotFile    ( 2sum.snap ())
InitialSnapshotFile  ( 2sum.init ())
)
```

The first five keys define the interval over which the problem is solved and other time related data. The keys `InitialTime`, `FinalTime`, and `InitialTimeStep` control the solution interval and the initial time step used to step through the solution points in this interval. By default, the time step is kept constant during the simulation. Thus, the `InitialTimeStep` key specifies the constant time step. Since you may not wish to generate output at every solution point, you are allowed to specify when reporting is to begin and the interval between reporting with `FirstReport` and `ReportCycle` respectively. Because we are working

a simple, algebraic problem here and want a single solution, we specify `FinalTime` to be the same as the `InitialTime` and `FirstReport` at time 0. The key `DiagnosticLevel` specifies the amount of intermediate output wanted in the run log file. This is discussed further in Section 12.5.

The remaining lines in the run-control file specify various files related to the problem. We have already discussed the 2sum.inp and 2sum.out files. Here we see that in the 2sum.run file you can specify where these files are located in your directory structure. In the above example, they are specified to reside in the current working directory. The other two keys, `InitialSnapshotFile` and `FinalSnapshotFile`, are discussed in Section 17. They specify the names of the snapshot files requested at the initial time and at the final time of the simulation run.

When the problem runs, summary output is displayed in the run log file and the principal output is written to the file called 2sum.out. For this problem the 2sum.out file contains:

```
3     Y     x     z
0     2     1     3
```

As with the input file, the first line gives the number of outputs, followed by the link names of each. The second line gives the time, followed by the result values for each output listed in the preceding line. As expected, adding 1 and 2 gives 3 !

2.2.2 Arbitrary Input/Output Designation

With SPARK, the problem can be changed without changing the model.

The preceding example showed the basic steps required to set up a *SPARK* problem. However, it did not show *SPARK*'s unique capabilities. One of these capabilities is that we can easily change which variables are input and which are output. That is, the problem can be changed without changing the model. For example, if we are interested instead in what *y* will be for specified values of *x* and *z*, we simply designate *z* as input and *y* as link:

```
/* Add 2 numbers together */
/*     2sum.pr            */
/*                        */
DECLARE  sum s;
LINK     x s.a    INPUT  REPORT;
LINK     y s.b           REPORT;
LINK     z s.c    INPUT  REPORT;
```

This results in the following directed graph for the *sum* problem with the re-arranged input/output designation.

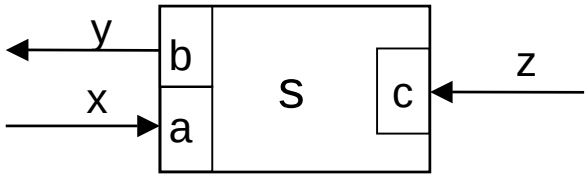

Figure 2-3: Directed graph for the new *sum* problem.

And, we must also change the input file to be:

```
2     x     z
0     1     3
```

The resulting output file, 2sum.out, contains:

```
3       z       x       y
0       3       1       2
```

Thus we see that *y* is calculated given *z* and *x*. Although shown here for a very simple problem with a single equation, this feature extends to more complex problems as well. The only requirement is that the model and the designated input variables must form a well-posed mathematical problem, i.e., one for which a solution exists.

2.3 PROBLEMS WITH SEVERAL OBJECTS

The previous examples were problems with a single equation and required only one *SPARK* object. Most real problems involve more than one equation and more than one object, thus raising the question of how objects are interconnected in *SPARK*. The following two examples show how this is done.

The problem we consider first is as follows:

$$x_1 + x_2 = x_5$$
$$x_3 + x_4 = x_6 \qquad (2.3)$$
$$x_5 + x_6 = x_7$$

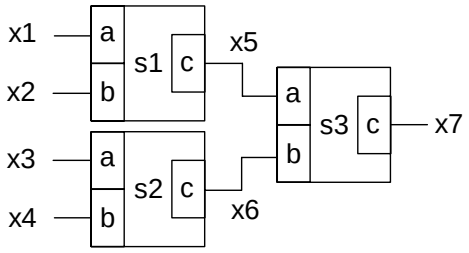

Figure 2-4: The *4sum* example

Obviously, each of these equations can be represented by an object of class **sum**. The diagram in Figure 2-4 shows how these objects would have to be interconnected to represent this problem.

The problem specification file for this problem contains the following code:

```
/* Add 4 numbers together */
/*     4sum.pr            */
DECLARE sum s1,s2,s3;
LINK  x1 s1.a        INPUT  REPORT;
LINK  x2 s1.b        INPUT  REPORT;
LINK  x3 s2.a        INPUT  REPORT;
LINK  x4 s2.b        INPUT  REPORT;
LINK  x5 s1.c, s3.a;
LINK  x6 s2.c, s3.b;
LINK  x7 s3.c               REPORT;
```

Observe that the `LINK` statement named *x5* connects the problem variable *x5* to the port **c** of *s1* and **a** of *s3*, demonstrating the basic object interconnection method of *SPARK*. Any number of object ports can be specified following the problem variable name, causing all to be equated to the single problem variable defined in the `LINK` statement and named after the `LINK` statement. The `LINK` and `DECLARE` statements (plus a few others yet to be discussed) form the *SPARK* language. The complete language is presented in reference form in Section 19.

Because there are four `INPUT` statements in 4sum.pr there must be a 4sum.inp file with values for the same four variables. This file is formatted as follows:

```
4       x1      x2      x3      x4
0       1       1       1       1
```

As before, the leading number in the first line, 4, is the number of inputs. It is followed by as many symbols, corresponding to input variables, as defined in 4sum.pr. The first number in the second line is the initial time, followed by values for each of the input variables.

The problem is built and executed using the same commands as for our *2sum* example. The results are placed in 4sum.out which is formatted like the input file:

```
5     x4     x3     x2     x1     x7
0     1      1      1      1      4
```

Several other files of interest are also produced when a *SPARK* problem is built and executed. First, various files with the extension *.log may appear in the workspace. As you might suspect, these contain any error or warning messages that may have been produced, as well as intermediate output and diagnostic from the numerical solution step.

Also produced is the equations file, e.g., 4sum.eqs. For complex problems exhibiting numerical difficulties, it is sometimes useful to examine this file because it contains the computation sequence determined by *SPARK* and used to solve the problem. For *4sum* this file contains:

```
Known variable(s) :
        x4                    INPUT
        x3                    INPUT
        x2                    INPUT
        x1                    INPUT


Component 0 :
   Solution sequence :
        x6                    = s2:sum__c( x3, x4 )
        x5                    = s1:sum__c( x1, x2 )
        x7                    = s3:sum__c( x5, x6 )
```

In this file, inputs are listed first, followed by a sequence of assignments to problem variables, each computed by a right-hand-side function reference. These functions represent the inverses of the underlying class equation. In this case there is only one component that contains three function references in a non-iterative sequence. Later, we will see that in more complex problems *SPARK* will break problems down into several components that can be solved independently in sequence. Note that here we use the word “component” in a graph theoretical sense, meaning a group of nodes and edges, i.e., equations and variables, that can be solved together; these have nothing to do with physical components. Some components are strongly connected, meaning that there are cycles in that part of the graph. The practical significance of strongly connected components is that the corresponding equations have to be solved simultaneously, by iteration until convergence. This is typically achieved using the Newton-Raphson solution method.

As with the single object example, we can use the same model to solve different problems by changing what is input and what is solved for. For example, suppose we want to specify *x5* and determine *x1*. The problem file is then:

```
/* Add 4 numbers together */
/*     4sum.pr            */
/*                        */
DECLARE sum s1, s2, s3;
LINK  x1 s1.a                 REPORT;
LINK  x2 s1.b          INPUT  REPORT;
LINK  x3 s2.a          INPUT  REPORT;
LINK  x4 s2.b          INPUT  REPORT;
LINK  x5 s1.c, s3.a    INPUT;
LINK  x6 s2.c, s3.b;
LINK  x7 s3.c                 REPORT;
```

A suitable input file is:

```
4       x5      x2      x3      x4
0       2       1       1       1
```

After building and executing, the resulting 4sum.out file is:

```
5       x4      x3      x2      x7      x1
0       1       1       1       4       1
```

And the equations file shows the solution sequence:

```
Known variable(s) :
        x4                      INPUT
        x3                      INPUT
        x5                      INPUT
        x2                      INPUT


Component 0 :
   Solution sequence :
        x6                      = s2:sum__c( x3, x4 )
        x7                      = s3:sum__c( x5, x6 )
        x1                      = s1:sum__a_or_b( x5, x2 )
```

Just as you might do, based on Figure 2-4, *SPARK* evaluates object *s2* followed by object *s3* in the "forward" direction yielding *x6* and *x7*, then evaluates object *s1* in the "reverse" direction to get *x1*.

2.4 PROBLEMS REQUIRING ITERATIVE SOLUTION

Up to this point all of our examples have been such that non-iterative solutions could be found. Each component could be solved using forward substitution of the unknowns in the sequence of equations. In more complex problems this may not be possible. For example, consider the set of equations below, in which c_1 and c_2 are given and x_1, x_2, x_3, and x_4 are to be determined.

$$\begin{aligned} x_1 + x_3 + x_2^2 + \sqrt{x_2} &= c_1 \\ x_2 &= x_1 e^{x_1} \\ x_1 x_4 + x_3 x_4 + x_4^3 &= c_2 \\ x_4 &= x_3 e^{-x_3} \end{aligned} \qquad (2.4)$$

This set of equations does not have a closed form solution and is very difficult to solve by any means. In fact, with some values of c_1 and c_2, it has no solution at all. However, with $c_1 = 3000$ and $c_2 = 1$ there is a solution and *SPARK* can easily find it.

The problem can be specified for *SPARK* exactly as for simpler ones. Figure 2-5 shows a *SPARK* diagram with objects and interconnections.

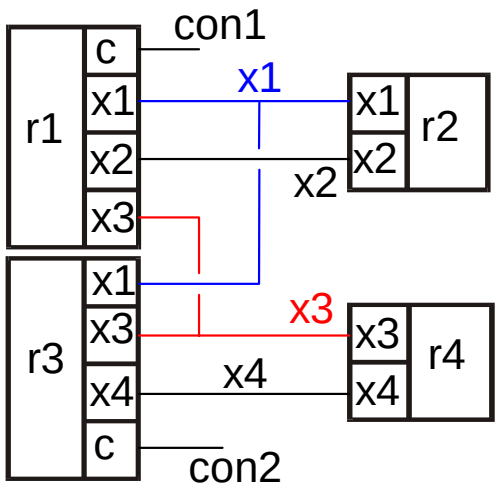

Figure 2-5: Four nonlinear equations

In this case we have used four objects, each representing one of the equations. We assume for the moment that there are classes **r1**, **r2**, **r3**, and **r4** representing the equations in the order given previously, presumed to have been defined and placed in the problem class directory.[2] The *SPARK* problem file can then be constructed as follows:

```
/*  Four nonlinear equations */
/*       example.pr          */
DECLARE r1  r1;
DECLARE r2  r2;
DECLARE r3  r3;
DECLARE r4  r4;

LINK  con1 r1.c                                  INPUT  REPORT;
LINK  con2 r3.c                                  INPUT  REPORT;
LINK  x1 r1.x1   MATCH_LEVEL = 0, r2.x1, r3.x1          REPORT;
LINK  x2 r1.x2, r2.x2                                   REPORT;
LINK  x3 r1.x3, r3.x3, r4.x3                            REPORT;
LINK  x4 r3.x4, r4.x4                                   REPORT;
```

The two constants, c_1 and c_2, in the equations are defined as input variables *con1* and *con2*. In these `INPUT` statements, the port variables representing the c_1 and c_2 constants are called **c** in the classes **r1** and **r3**. Similarly, in the `LINK` statements it is evident that the other port variables have the same names as the corresponding problem variables. Usually, in the interest of code reuse, it is better to define a generic class using local names for port variables, as we have done in the earlier examples. Here, however, where it is unlikely that we will have need for other instances of these rather specialized objects, it would introduce unnecessary confusion to employ different port and problem variable names. Hence the *x1* problem variable is linked to the **x1** port variable of all objects in which it occurs, i.e., *r1*, *r2* and *r3*.

A new *SPARK* language keyword, `MATCH_LEVEL`, is used in this problem. The purpose of this keyword is to provide a hint to *SPARK* on how to match certain variables to certain equations. Here, by placing the `MATCH_LEVEL = 0` after the **r1** port connection for **x1**, we are discouraging *SPARK* from using the *r1* object, i.e., the first equation, to calculate *x1*. Although most often *SPARK* can do without such hints, there may be times when you have particular insights into the numerical properties of the problem, and the `MATCH_LEVEL` keyword provides one mechanism for capitalizing on this knowledge. For example, experience with the current problem indicated that the above `MATCH_LEVEL` restriction leads to a better solution sequence. Unfortunately, it is not always easy to discover appropriate matching preferences, but when you do develop the insight for a particular problem it is important to be able to control *SPARK* in this manner. This subject is discussed further in Section 12.3.

The results of running *SPARK* on the problem so described, with values of 3000 and 1 for the constants *con1* and *con2*, respectively, are shown below:

```
  6     con2       con1       x4          x1          x2          x3
  0     1          3000       0.288576    2.927303    54.67379    0.4547163
```

Naturally, the values reported for x_1 through x_4 satisfy the given equations.

The equations file, example.eqs, shows how *SPARK* arrived at these answers:[3]

```
Known variable(s) :
        con2                INPUT
```

[2] We will see how to define SPARK object classes in Section 2.4.

[3] As is often the case for nonlinear problems, this example has multiple solutions. The solution found will depend upon the starting point in the iterative solution process.

```
        con1                    INPUT


Component 0 :
   Break variable(s) :
        x3                      PREDICT_FROM_LINK = x3

   Solution sequence :
        x4                      = r4:r4_x4( x3 )
        x1                      = r3:r3_x1( x3, x4, con2 )
        x2                      = r2:r2_x2( x1 )
[BREAK] x3                      = r1:r1_x3( x1, x2, con1 )
```

We see there is a single component (called "`Component 0`") in the solution, meaning that this problem does not allow partitioning. Within this single component, the function *r4_x4(x3)* represents object *r4*, i.e., the fourth equation in (2.4), solved for *x4* in terms of *x3*. The value returned by the function is assigned to the *x4* problem variable. Similarly, *r3_x1(x3, x4, con2)* represents object *r3*, i.e., the third equation, solved for *x1*, *r2_x2(x1)* is *r2* solved for *x2*, and finally *r1_x3(x1, x2, con1)* is *r1* solved for *x3*. It is apparent that these assignments form a cycle, i.e., *x1* must be known to get *x3*, but *x3* must be known to get *x1*. That is, the component is **strongly connected**. Recognizing this, *SPARK* has selected *x3* to break the cycle, i.e., a value of *x3* is guessed to start an iterative solution process. Thus after evaluating *r1_x3* (using the guessed value of *x3* to get *x4* and then *x1* and *x2*) *SPARK* will use a numerical method for estimating a new value of *x3* and repeat the calculations from the first assignment. This will continue until the predicted and calculated values of *x3* agree to within the *SPARK* precision, which defaults to 10^{-6}. By default, the solution is computed with the Newton-Raphson method. If convergence is not achieved, alternate methods can be tried. Usually, convergence is obtained with the Newton-Raphson method.

The above functions are based on the respective object class equations. By chance, the function *r4_x4* happens to be the way the *r4* equation was originally expressed, i.e., as a formula for *x4* in terms of *x3*. However, the function *r3_x1* is the *r3* object class rearranged symbolically, i.e.,

$$x_1 = \frac{\left(c_2 - x_3 x_4 - x_4^{\,3}\right)}{x_4} \tag{2.5}$$

This is called an **inverse** of the object. Part of the task of developing a *SPARK* class is performing these symbolic inversions of the given equations and embedding them in *C++* functions. This is discussed in Section 3.2

2.5 ITERATIVE SOLUTION AND BREAK VARIABLES

As we have noted in the previous example, systems of equations often have to be solved iteratively. In *SPARK*, this can be true even if the equations are all linear, because no specific test is done for linearity. Usually, you need not be concerned with the iterative process, so we will not go into detail here. However, a general awareness of the methods used is helpful if solution difficulties are encountered.

First, in the problem setup phase, *SPARK* determines if iteration is required by detecting cycles in the problem graph. If cycles are detected, a graph algorithm is used to find a small set of variables (nodes in the graph) that "cut" the cycles. The associated problem variables, called **break variables,** are placed in a vector to act as the unknown vector *x* in a multi-dimensional Newton-Raphson solution scheme. The **residual functions** that are forced to zero in the Newton-Raphson process are of the form

$$F(x) = f(x) - x \tag{2.6}$$

where x is the vector of break variables, and $f(x)$ represents the directed acyclic graph formed when the original problem graph is cut at the cut-set vertices. In other words, the current solution estimate, *x*, is applied to the graph, producing $f(x)$, from which the original estimate is subtracted. At the solution, the residual functions is equal to zero.

$$F(x) \equiv 0 \tag{2.7}$$

The Jacobian matrix for the residual function is defined as

$$J = \frac{\partial F}{\partial x} \tag{2.8}$$

In each Newton-Raphson iteration the next estimate x_{next} is calculated by solving the linear set

$$J\Delta x = -F(x) \tag{2.9}$$

for Δx, then calculating

$$x_{next} = x + \Delta x \tag{2.10}$$

The solution of the linear set, Equation (2.9), is carried out with Gaussian elimination, LU decomposition, or similar method. Note that the size of the Jacobian matrix is the size of the cut set, so this solution can be much more efficient than if we had not attempted to minimize the cut set.

Normally, this process converges to the solution quite rapidly (quadratically). However, it is well known that the Newton-Raphson process, like all methods for solving general sets of nonlinear equations, can fail to converge under certain circumstances. Failure occurs when the residual functions have particular kinds of non-linearities and the starting values are not sufficiently close to the actual solution. Thus starting values are important.

In *SPARK*, we refer to the process of selecting a starting value for the iteration process as "prediction." By default, the prediction for solution at a particular time step is the final solution value for the same variable at the previous time step. This can be changed by use of the `PREDICT_FROM_LINK=linkFrom` keyword in the corresponding `LINK` statement. In this case, the value of the `linkFrom` link is used as the predictor.

Note that at the initial solution when the time is equal to `InitialTime`, there is no proper "previous time step value." In this case, if there is no `PREDICT_FROM_LINK=linkFrom`, *SPARK* will use the default value for the break variable as the initial predictor. Since default values determined in this way are not appropriate for every variable, they may not be very close to the solution value. Therefore it is best to provide **initial predictors** via input files. This issue is discussed further in Section 7.2.

2.6 WELL-POSED PROBLEMS

In Section 2.2.2 we saw that *SPARK* allows us to change which problem variables are input and which are to be solved for without changing the underlying model. This flexibility is the result of specifying object models without a priori specification of inputs and outputs (Sahlin and Sowell 1989). Thus we were able to solve for x_5, x_6, and x_7 in the example Equation (2.3) given x_1 through x_4, or by a simple change of `INPUT` and `LINK` designations solve for x_1, x_6, and x_7 given x_2, x_3, x_4, and x_5.

It would be grand if we could say that this selection of the input and output sets was completely arbitrary. For example, in the example of Section 2.3, Equation (2.3), there are three equations (objects) and seven variables, so one might hope that any set of four inputs could be used to determine the remaining three variables. However, we are constrained by what is mathematically possible. In many problems there are sets of inputs that will not define a problem that has a solution. For example, if we specified x_2, x_3, x_4, and x_6

it is impossible to determine a solution. From Figure 2-4, we see that if x_3 and x_4 are both specified then x_6 cannot be specified. Moreover, there is no way to determine x_1, x_5, and x_7 given only x_6. Mathematically, a problem is said to be **well-posed** if it admits a solution. Thus with this input set we have an ill-posed problem.

Naturally, *SPARK* has no ability to solve ill-posed problems; however, in this case, *SPARK* can immediately determine that the problem is not well-posed. Specifically, it discovers that there is no possible **matching** of equations and variables. Other forms of ill-posedness cannot be discovered until a numerical solution is attempted and, in such cases, a lack of convergence will be reported. Unfortunately, however, lack of convergence also may be the result of other numerical problems, such as improper starting values, so we cannot always conclude that this means ill-posedness. Problems of this nature are all too familiar to those who routinely work with nonlinear systems of equations. Often, insights afforded by knowledge of the physical problem under analysis suggest ways to fix the numerical problem. In seeking to resolve these difficulties, we should be motivated by the realization that proper mathematical models of physical systems are well-posed. Otherwise, the physical system could not behave in the observed way.

In summary, *SPARK* offers a method for specifying and solving sets of equations in a computationally efficient way, provided solution is possible. But it should be no surprise that it cannot solve insoluble problems, and numerical difficulties may be encountered as they would be in other solution methods.

3 CREATING *SINGLE-VALUED* ATOMIC CLASSES

In Sections 2.2 and 2.3, the examples have so far made use of existing *SPARK* classes that implement single-valued inverses, i.e., calculating the value for a single port. In practice, it is often necessary to create new classes to meet special needs, as shown in Section 2.4. This can be done either by hand, with symbolic tools such as the *SPARK* symbolic solver, or third-party tools like *Maple*, *Mathematica* or *MACSYMA*. Here we present the manual process of creating a single-valued atomic class which will allow you to better understand the use of the symbolic tools as discussed in Section 3.3. The process of creating a multi-valued atomic class will be discussed in Section 4.

3.1 CLASS DEFINITION

Creating a *SPARK* atomic class is a two step process. First, you must create the class definition. Second, the functions required by the class definition[4] must be expressed in *C++* following the *SPARK* function protocol. The class definition and the supporting *C++* callback functions are stored in the same file with a .cc extension. These steps are demonstrated below for the **sum** atomic class.

```
#ifdef SPARK_PARSER

PORT a  "Summand 1" ;
PORT b  "Summand 2" ;
PORT c  "Sum" ;

EQUATIONS {
   c = a + b ;
}

FUNCTIONS {
   a = sum__a_or_b( c, b ) ;
   b = sum__a_or_b( c, a ) ;
   c = sum__c( a, b ) ;
}

#endif /* SPARK_PARSER */
#include "spark.h"

EVALUATE( sum__a_or_b )
{
    ARGDEF(0, c);
    ARGDEF(1, b);
    double a_or_b ;

    a_or_b = c - b;
    RETURN( a_or_b ) ;
}
EVALUATE( sum__c )
{
    ARGDEF(0, a);
    ARGDEF(1, b);
    double c;
```

[4] They are referred to as the callback functions since they implement a callback mechanism with the solver.

```
    c = a + b;
    RETURN( c ) ;
}
```

As shown above, it is customary to begin a class with comments to describe what it does. After the comment header comes the body of the class definition. This is placed within *C*-style `#ifdef SPARK_PARSER` and `#endif` so the file can be processed both by the *SPARK* parser and the *C++* compiler.

3.1.1 The PORT Statement

The first part of the class definition is a list of the ports. It is through these ports that objects of the class communicate with other objects. Although the `PORT` statement has additional optional clauses (See Section 19.10), the only requirement is the name of the port variable. Here, we also provide a description string that is used for error reporting.

The port variable name can be arbitrarily chosen and of any length and is placed following the `PORT` keyword. Note that throughout the *SPARK* language user selected names are case sensitive; however, keywords of the language are not. Thus, either `port` or `PORT` will do, but `a` and `A` are considered different `PORT` names. Like all *SPARK* statements, the `PORT` statement can span multiple lines if necessary. Each `PORT` statement ends with a semicolon.

3.1.2 The EQUATIONS Statement

After the `PORT` declaration, the equation for the class can be given in the optional `EQUATIONS` block. Although this *SPARK* atomic class presently has a single equation, the possibility of multiple equations is allowed for with the compound statement using braces, `EQUATIONS {…}`. [5]

3.1.3 The FUNCTIONS Statement

Following the equations is the `FUNCTIONS {…}` compound statement that defines the set of inverses assigned to each mutually exclusive set of ports between the braces. An inverse consists of a set of **target** ports followed by the `=` sign and a list of comma-separated callback functions, each prefixed by a callback keyword. A callback function is specified with its name followed by the list of the argument ports using parenthesis. The **argument** ports are the ports whose values are needed to implement the behavior of the callback. Each inverse statement ends with a semicolon. The following code snippet shows the structure of the `FUNCTIONS` statement.

```
FUNCTIONS {
    port1 = <callback-keyword1> inverse1_callback1( port2, port3, …),
            <callback-keyword2> inverse1_callback2( port2, port3, port4,
    …)
    ;
    port2 = <callback-keyword1> inverse2_callback1( port1, …),
            <callback-keyword2> inverse2_callback2( port1, port4, …)
            …
    ;
}
```

The list of callback keywords and the behavior of each callback function is explained in Section 9. The callback function responsible for calculating the values of the target port variable(s) assigned to it is the `EVALUATE` callback function. It is the only callback function for which the callback keyword may be omitted.

[5] The equations block is optional since *SPARK* currently does not process it. Future releases may automatically generate the *C++* functions based on the equation block.

Since any inverse defines the EVALUATE callback function it is customary to refer to each inverse with the name of the associated EVALUATE callback function. Here the atomic class **sum** defines three inverses:

- the inverse `sum__a_or_b` assigned to the port **a**;
- the inverse `sum__a_or_b` assigned to the port **b**; and
- the inverse `sum__c` assigned to the port **c**.

Note that each inverse is assigned to a different port and that it defines only the EVALUATE callback function. The inverse assigned to the port **a** is different than the inverse to the port **b** although they both rely on the same callback function `sum__a_or_b`.

An inverse that calculates the value of only one port variable is referred to as a single-valued inverse, whereas an inverse that calculates simultaneously the values for more than one port variable is referred to as a multi-valued inverse (See Section 4).

In the atomic class **sum** we define three single-valued inverses for calculating each of the three port variables. Usually, defining an inverse for each port variable is the best practice, since it allows *SPARK* greatest flexibility and efficiency in devising a solution strategy for various problems in which the class might be used. That is, some problems may require **c** to be determined in terms of **a** and **b**, while in others it may be preferred to calculate **b** given **a** and **c**. As we shall see below, each inverse is a "mathematical" inverse function of the object equation.

For complex equations, some inverses may be difficult or impossible to obtain. Or, it may be that special knowledge about the problem under investigation suggests that a particular inverse should not be used, because, for example, it might lead to numerical difficulties. For these reasons, *SPARK* allows you to omit unavailable or unwanted inverses. For example, we could simply omit the function for calculating **a** from the **sum** class. Should the need to calculate **c** from **a** and **b** then arise in some problem using the class, *SPARK* would have to perform the calculation iteratively.

3.2 INVERSE FUNCTIONS DEFINITION

After the class definition comes the definition of the inverse functions. These functions, supporting the *SPARK* class definitions, are expressed as *C++* free functions implementing the **callback functions** (See Section 9). Although some familiarity with *C++* would be helpful here, you should be able to understand the discussion with background in any similar language.

3.2.1 Basic Structure of a Single-Valued EVALUATE Callback

The basic structure of the EVALUATE callback function in a *SPARK* atomic class is:

```
EVALUATE( funct_name )
{
   // Code for calculating the result from the arguments,
   // returned as a double using the RETURN preprocessor macro.
   double result;
   …
   RETURN( result );
}
```

In order to make the definition of the callback functions easier to the user, *C* preprocessor macros that hide the implementation details of argument passing as well as the function prototype are defined in the header file spark.h. With these preprocessor macros we can write the body for the `sum__c` callback function as follows:

```
EVALUATE( sum__c )
```

```
{
    ARGDEF(0, a);
    ARGDEF(1, b);
    double c;

    c = a + b;
    RETURN( c ) ;
}
```

3.2.2 Defining the C++ Callback Function

The code

```
   EVALUATE( sum__c )
```

declares the *C++* function `sum__c` as an `EVALUATE` callback. The `EVALUATE` callback is responsible for calculating the value of the port variable assigned to the inverse in the `FUNCTIONS {…}` statement. In this case, the `EVALUATE` callback of the `sum__c` inverse calculates the value of the target port **c** from the values of the argument ports **a** and **b**. Other callback functions can be specified for a *SPARK* inverse to implement other operations beside evaluating the value of the target port. However, the `EVALUATE` callback is the only function that must always be specified for any inverse.[6]

The macro preprocessor `EVALUATE` expands to the *C++* function prototype expected for a `EVALUATE` callback with the proper argument list.

3.2.3 Defining the Argument Variables

The code

```
   ARGDEF(0, a);
```

declares `a` as the argument port passed in the first position (index 0) to the callback function `sum__c` in the `FUNCTIONS { }` statement for this inverse. Note that indexing in the argument list is zero-based as is customary in *C++*. Similarly, the code

```
   ARGDEF(1, b);
```

declares `b` as the argument port passed in the second position (index 1) to the callback function `sum__c`.

The *SPARK* solver implements the argument variables as instances of the class `TArgument`. Thus, the macro preprocessor `ARGDEF` declares the variables `a` and `b` as instances of this *C++* class.

3.2.4 Calculating the Result Value

The code

```
   double result = a + b;
```

calculates the sum of the argument ports **a** and **b** to be assigned to the target port **c**. This statement implements the mathematical relation expected by this inverse using the `TArgument` instances `a` and `b`.

The *C++* variables named `a` and `b` are directly used in the arithmetic expression `a + b` to compute the sum of the two associated argument ports as a `double` value. This is possible because the class `TArgument`

[6] The default *SPARK* atomic class describes an equation that returns the value(s) of the target port(s) that is/are assigned to each inverse. However, there are other types of atomic classes (See Section 8.6) that are not required to return value(s). Therefore, these classes will not define the `EVALUATE` callback.

behaves as a numerical value by overloading the `operator double()` method. Therefore, the *C++* variables `a` and `b` return their respective current numerical values where they appear in the expression.

Each instance of the class `TArgument` also stores information about the other properties of a *SPARK* variable, such as its name, units, minimum value, maximum value, and initial value. The htm/chm tutorial *SPARK Atomic Class API* should be consulted for more information on the class `TArgument` and how to use its methods.

3.2.5 Returning the Result Value

Finally, the `RETURN` preprocessor macro takes care of returning the calculated value `result` to the variable connected to the target port **c** for this single-valued inverse `sum__c`. This ensure the proper data flow across the set of the unknown variables in the problem.

SPARK functions can be as simple as the above example, or quite complicated. The full expressive power of *C++* is allowed. Also, code for existing models can be integrated by means of a function call. Furthermore, by following the rules for mixed language programming in your environment, the referenced functions can be in *FORTRAN*, *Pascal*, or *Assembly* language. The principal requirement is that **the EVALUATE callback function returns the calculated value for the associated target variable**. Perusal of some of the classes in the *SPARK* globalclass and hvactk\class directories may be beneficial before beginning development of complex classes of your own.

3.3 SYMBOLIC PROCESSING

As seen in earlier examples, *SPARK* atomic classes are constructed from equations. While these classes can be constructed manually, the process can be time consuming and tedious. First, the equation must be solved for all (or most) of its variables, one at a time. For example, if the equation is the ideal gas relationship, $pv = nRT$, we need to do the algebra to get the following formulas:

$$\begin{aligned} p &= nRT / v \\ v &= nRT / p \\ n &= pv / RT \\ R &= pv / nT \\ T &= pv / nR \end{aligned} \tag{3.1}$$

These are called **inverses** of the original equation. Then, for each inverse we must construct a *C++* function that evaluates the right hand side and returns the resulting value. Finally, all of these functions must be incorporated in a *SPARK* atomic class representing the ideal gas law, following the syntax shown in the earlier examples (Section 3.1).

Fortunately, these tasks can be automated using symbolic processing (also called computer algebra) tools. *SPARK* provides a program called *sparksym* that fills this need. With it you can generate all symbolic inverses of an algebraic equation, generate *C++* functions implementing these single-valued inverses, or create the complete *SPARK* atomic class.

Actually, *sparksym* is an interface to third-party symbolic programs. Currently, it can use either *Mathomatic*, *Maple*, *Mathematica* or *MACSYMA*, as selected by a command line option. A subset of the *Mathomatic*

program is integrated in *sparksym*, so that option is always available.[7] If *Maple*, *Mathematica* or *MACSYMA* are detected on your machine when *SPARK* is installed, or if you install them later and take steps to link them to *SPARK*, you can select one as an alternative symbolic engine for *sparksym*. *Maple*, *Mathematica* and *MACSYMA* are more powerful than *Mathomatic*, allowing more complex equations to be handled.

3.3.1 Simple Symbolic Processing

Command-line usage of *sparksym* is with the command:

```
sparksym –engine -option [name] "equation" [target] [outFile] <enter>
```

where:

`engine`	=	`O` (*Mathomatic*), `P` (*Maple*), `E` (*Mathematica*), `S` (*MACSYMA*)
`option`	=	`i` (single inverse), `a` (all inverses), `f` (function), `c` (class)
`name`	=	Name for function of class (used only with option f or c)
`equation`	=	An equation of the form `<expression>=<expression>` (enclose in double quotes if spaces occur)
`target`	=	The variable to be solved for (used only for options `i` and `f`)
`outFile`	=	Optional file for the result

3.3.2 Generating an Inverse

For example, to generate the inverse equation for *T* using the ideal gas law, with output to the screen:

```
sparksym –O -i "p*v = n*R*T"  T <enter>
Inverse:
 T = p*v/n/R
```

Or to create the *SPARK* **idealGasLaw** atomic class, with results written to idealGasLaw.cc:

```
sparksym –O -c idealGasLaw "p*v=n*R*T" idealGasLaw.cc <enter>
```

The class generated is directly usable, but perhaps not as complete as you may wish. For example, the ports are all assigned a description which is the same as the port name, units are [-] (i.e., unspecified), and the `INIT`, `MIN`, and `MAX` values are set at 1, -100000, and 100000 respectively. You can edit the output file to give more appropriate values for these items if you wish.

3.3.3 Caveats

You are advised to carefully check all symbolic results, since computer algebra software often gives unexpected results, sometimes simply wrong.

Sparksym using the *Mathomatic* option is not as robust as a full-featured symbolic package, although it may meet many of your needs. With it, you are limited to expressions using the operators +, -, *, /, and ^ (exponentiation). It will fail quickly if it cannot easily invert the equation for the desired variable. Note that the atomic class generated with the `-c` option will have functions for each variable in the equation, whether or not an explicit inverse was found for it. Variables for which it could not find an explicit inverse use an implicit inverse as in Section 12.2. You may wish to edit the implicit functions, as discussed in the same section, to improve numerical stability.

[7] The *sparksym* executable provided with *SPARK* does not give you the full capability of *Mathomatic*. You can download the *DOS* shareware program from http://www.lightlink.com/george2/. Among other features, it is capable of symbolic elimination of variables and equations in sets of equations; sometimes this feature can be used to help develop efficient *SPARK* classes.

With the *Maple* option, practically any equation can be handled, including various mathematical functions. Additionally, it will sometimes find multiple inverses. In this case all inverses are written in the generated functions, with all but one commented out. Therefore it is a good idea to examine the generated class to see that the wanted inverse is being used.

4 CREATING MULTI-VALUED ATOMIC CLASSES

One limitation of the concept of the single-valued atomic class presented in Section 3 is that only a single result can be communicated to the rest of the problem, even though many variables may be determined in the process. In order to deal with situations like that, it is possible to define multi-valued atomic classes that will determine the values of more than one variable simultaneously. This section describes the process of defining the inverse of a multi-valued atomic class.

4.1 MOTIVATION

When writing the model classes situations also arise where a developer or analyst wishes to use a model expressed in an algorithmic language within a *SPARK* model (Sowell 2003). This comes up when there is an existing, trusted model 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* model through the definition of the equivalent single-valued atomic classes.

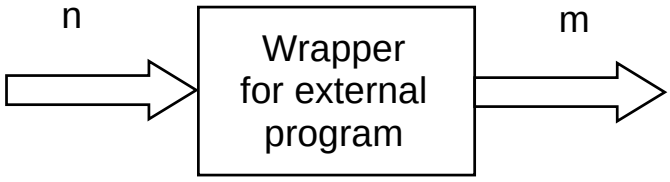

Figure 4-1: Diagram representing a wrapper multi-valued class.

Additionally, sometimes there are small sets of equations within a system that are numerically problematic for any global iterative solution scheme, 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* approach of breaking subsystems 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 using multi-valued atomic objects, rather than the normal single-valued atomic objects.

Following is a non-exhaustive list of possible applications for multi-valued atomic classes:

- Compute values for a set of target variables at once, thus potentially saving duplicate intermediate calculations used to calculate the value for each individual target variable.
- Implement the symbolic solution of a set of equations instead of relying on the *SPARK* solver to find the numerical solution.
- Implement rule-based, multi-dimensional control algorithms as *C++* code.
- Specify a wrapper atomic class around a third-party program which describes a directed set of equations with fixed inputs and outputs as shown in Figure 4-1. E.g.,
    - Integrate a legacy-code model expressed in any procedural language.
    - Couple *SPARK* with other programs such as a CFD code.
    - Operate a *SPARK* model in real-time with embedded digital controllers and sensors.

- Specify a wrapper atomic class around another *SPARK* problem to embed inside a master *SPARK* problem. It is also possible to model discrete states for the atomic class implemented through different embedded problems that describe the change in formulation of the underlying equations corresponding to each discrete state.

4.2 LIMITATIONS

When writing a multi-valued atomic class you become responsible for devising an algorithm for the EVALUATE callback function that calculates the values of the target ports as a multi-dimensional function, thereby bypassing one of *SPARK's* most unique capabilities. If the underlying equations of the multi-valued inverse are described as a nonlinear problem in residual form, then the appropriate solution algorithm[8] must be implemented in the body of the callback function.

It is of course possible to define a separate *SPARK* problem that solves this set of nonlinear equations and embed the resulting problem in the multi-valued inverse using the *SPARK* Problem Driver API documented in separate html/chm help files. Thus, the solution algorithm for the subproblem is devised automatically by *SPARK* and your task consists only in embedding the subproblem in the multi-valued atomic class.

Another limitation of the current implementation of the multi-valued inverse mechanism stems from the matching algorithm executed in the *setupcpp* program. **Only one multi-valued inverse can be specified per atomic class.** Thus, the multi-valued objects represented as directed objects in the computational graph force the links connected to the target ports to be matched with their unique multi-valued inverse. Future versions of *SPARK* might be capable of handling more than one multi-valued inverse per atomic class as long as each inverse is assigned to a the set of mutually exclusive target ports.

4.3 CLASS DEFINITION

As an example of a multi-valued atomic class we use the root2.cc atomic class that is part of the global classes stored in the globalclass subdirectory. This atomic class calculates the roots of a 2nd order polynomial described through its coefficients. This is a clear case where using a single multi-valued inverse to calculate both roots simultaneously is more efficient because it allows to reuse the intermediate value for the discriminant in the calculation of each possibly distinct root.

```
// root2.cc
// Multi-valued object that returns the 2 roots of a second order
// polynomial
//     a*x^2 + b*x + c = 0
////////////////////////////////////////////////////////////////////

#ifdef SPARK_PARSER

PORT a;
PORT b;
PORT c;
PORT root_plus;
PORT root_minus;
PORT discriminant;


EQUATIONS {
    a*x^2 + b*x + c = 0;
```

[8] The Newton-Raphson method is usually used to solve such a set of nonlinear equations.

```
}

FUNCTIONS {
   root_plus, root_minus, discriminant = root2__mroot2( a, b, c )
     ;
}

#endif /* SPARK_PARSER */
#include "spark.h"

EVALUATE( root2__mroot2 )
{
    ARGUMENT( 0, a );
    ARGUMENT( 1, b );
    ARGUMENT( 2, c );

    TARGET( 0, root_plus );
    TARGET( 1, root_minus );
    TARGET( 2, discriminant );

    double discriminantx = b*b - 4.0*a*c;

    if (discriminantx < 0.0) { // Atomic class error
        REQUEST__ABORT( "Cannot compute complex roots." );
    }

    double square_discriminant = sqrt( discriminantx );

    root_plus  = (-b + square_discriminant)/(2.0*a);
    root_minus = (-b - square_discriminant)/(2.0*a);
    discriminant = discriminantx ;
}
```

Since only a single inverse is allowed for a multi-valued atomic class, the data flow through the class is automatically directed from the argument ports to the target ports as shown in Figure 4-2. Using multi-valued classes therefore constrains the matching algorithm performed in the *setupcpp* program and might lead to incomplete matching.

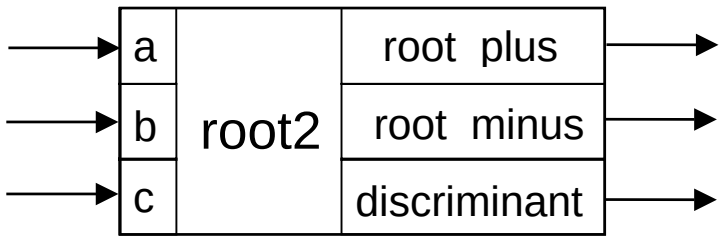

Figure 4-2: Directed graph representing the **root2** multi-valued atomic class.

4.3.1 The PORT Statement

The first part of the class definition is a list of the ports. The root2.cc atomic class defines six ports:

- one for each coefficient **a**, **b** and **c**;

- one for each possibly distinct root, **root_plus** and **root_minus**; and
- one port to expose the value of the discriminant, **discriminant**.

Note that the port **discriminant** could be defined with the keyword NOERR to indicate that it is not required that this port be connected to any link variables in the problem definition, thus making it an optional port.

4.3.2 The EQUATIONS Statement

After the PORT declaration, the equation for the class is given in the optional EQUATIONS block.

4.3.3 The FUNCTIONS Statement

Following the equations is the FUNCTIONS {…} compound statement that defines the unique multi-valued inverse assigned to the target ports **root_plus**, **root_minus** and **discriminant** between the braces. As with single-valued inverses, the list of target ports is specified on the left hand-side of the = sign. This inverse defines only the EVALUATE callback function named root2__mroot2, which depends on the three polynomial coefficients defined as the ports **a**, **b** and **c**.

4.4 INVERSE FUNCTION DEFINITION

4.4.1 Defining the C++ Callback Function

After the class definition comes the definition of the inverse functions. The code

```
EVALUATE( root2__mroot2 )
```

declares the *C++* function root2__mroot2 as an EVALUATE callback which calculates the values of the target ports **root_plus**, **root_minus** and **discriminant** from the values of the argument ports **a** , **b** and **c**.

4.4.2 Defining the Argument Variables

The code

```
ARGUMENT( 0, a );
```

declares a as the argument port passed in the first position (index 0) to the callback function root2__mroot2 in the FUNCTIONS { } statement for this inverse. This declaration using the ARGUMENT macro is exactly equivalent to using the ARGDEF macro preprocessor as shown in Section 3.2.

Similarly, the code

```
ARGUMENT( 1, b );
ARGUMENT( 2, c );
```

declares b and c as the argument ports passed in the second (index 1) and third (index 2) positions to the callback function root2__mroot2.

4.4.3 Defining the Target Variables

The code

```
TARGET( 0, root_plus );
```

declares root_plus as the target port defined in the first position (index 0) in the list of target ports specified for the inverse in the FUNCTIONS {…} statement. The *SPARK* solver implements the target variables as instances of the class TTarget. Thus, the macro preprocessor TARGET declares the variable

`root_plus` as an instance of this *C++* class. The tutorial *SPARK Atomic Class API* should be consulted for more information on the class `TTarget` and how to use its methods.

Similarly, the code

```
TARGET( 1, root_minus );
TARGET( 2, discriminant );
```

declares the instances `root_minus` and `discriminant` of the class `TTarget` for the target ports specified in the second (index 1) and third (index 2) positions.

4.4.4 Calculating the Result Values

The last part of the *C++* code of the callback function deals with calculating the values for each target ports and assigning the results to each of the associated `TTarget` instances. Here the full expression of the *C++* programming language can be used to derive the results from the values of the `TArgument` instances.

The distinct real roots of a 2nd-order polynomial expressed as

$$a \cdot x^2 + b \cdot x + c = 0 \tag{4.1}$$

are obtained with the following equations when the discriminant Δ is strictly positive:

$$\begin{cases} \Delta = b^2 - 4 \cdot a \cdot c \\ x_+ = \dfrac{-b + \sqrt{\Delta}}{2 \cdot a} \\ x_- = \dfrac{-b - \sqrt{\Delta}}{2 \cdot a} \end{cases} \tag{4.2}$$

When the discriminant is equal to zero, then there is a real double root instead of the two distinct real roots, $x_{root} = x_+ = x_-$. When the discriminant is negative, the two roots are no longer real but conjugate complex numbers. Since *SPARK* does not have native support for complex numbers, we will not treat this case in our implementation of the root2.cc atomic class.

The code

```
double discriminantx = b*b - 4.0*a*c;

if (discriminantx < 0.0) { // Atomic class error
    REQUEST__ABORT( "Cannot compute complex roots." );
}
```

calculates the value of the discriminant Δ using the previous equation and stores it in the temporary `double` variable `discriminantx`. Note that we use the `TArgument` instances `a`, `b` and `c` directly in the *C++* code that implements the discriminant equation. Indeed, thanks to the overloaded `double` operator, a `TArgument` instance returns its current value as a `double` value whenever mentioned in a *C++* expression that expects a `double` value. It is also possible to use any other methods of the `TArgument` class in the code.

The following `if {…}` statement detects when the discriminant is negative and stops the simulation by sending an `ABORT` request to the solver. The request is implemented using the `REQUEST__ABORT` preprocessor macro that takes a `const char*` string as an argument to identify the context of the request at runtime. See Section 10 for more information on the request mechanism.

The code

```
    double square_discriminant = sqrt( discriminantx );
```

calculates the square root of the discriminant using the *C* library function `sqrt` defined in `<cmath>`.

4.4.5 Returning the Result Values

Finally, the code

```
    root_plus  = (-b + square_discriminant)/(2.0*a);
    root_minus = (-b - square_discriminant)/(2.0*a);
    discriminant = discriminantx ;
```

assigns the result values to each `TTarget` instance. This statement implements the mathematical relations shown in Equation (4.2) to compute the real distinct roots x_+ and x_-. Again, we can freely write code that mixes the `TArgument` instances and the locally defined, temporary variables defined as `double`.

Unlike in our example of a single-valued inverse in Section 3.2, we can no longer rely on the convenient preprocessor macro `RETURN` to assign the result values to the target variables. Indeed, the `RETURN` macro assumes that there is only one target port assigned to the inverse, therefore always returning the result value for the first (and only!) target port.

In the case of a multi-valued inverse, you have to explicitly write the assignment statements for each `TTarget` variable before returning from the `EVALUATE` callback function. This is achieved by using the operator = of the `TTarget` class, whereby on the left hand-side of the `=` sign you write the name of the `TTarget` instance and on the right hand-side the `double` result value or a *C++* expression that can be evaluated as a `double` value.

For example, the code

```
    root_plus  = (-b + square_discriminant)/(2.0*a);
```

assigns the value of the mathematical relation for the root x_+ to the `TTarget` instance `root_plus`. A similar assignment statement follows for the `TTarget` instance `root_minus` using the corresponding mathematical relation for the root x_-.

The last statement assigns the value of the temporary `double` variable `discriminantx` to the `TTarget` instance `discriminant`. This line simply copies the value of the local variable to the internal data structure representing the *SPARK* variable. You might wonder why we did not directly use the `TTarget` instance `discriminant` to hold the value computed for `discriminantx` in the first place. The reason why this is not allowed originates in the graph-theoretical analysis performed in the *setupcpp* program.

In order to ensure the correct variable dependency between target and argument ports across all the atomic objects defined in the *SPARK* problem, the value of a target port cannot be used in the computation of the inverse assigned to the target ports unless the same port is also specified as an argument port. Essentially, the argument ports provide read-only access to their current values through the overloaded operator `double`, whereas the target ports provide write-only access to their current numerical values through the overloaded operator `=`. If you need to access the current value of a target port in the callback function, then you must also declare the port in the argument list of the corresponding callback in the `FUNCTIONS {…}` statement. Thus, the inverse appears to depend unequivocally on the value of one of its target port(s), which forces the variable connected to this target port to be a break variable.

Were it possible to access the current value of a target port when computing the value of the same target port, it would create a hidden dependency between the target variable and the inverse it is matched with, resulting

in an algebraic loop that would go unnoticed during the graph-theoretic analysis. Clearly, this situation would produce a wrong solution sequence that no longer accounts for the topology actually described by the underlying equations.

This explains why we had to use a temporary variable `discriminantx` to hold the value of the discriminant so that this value could also be reused in the computation of the two roots. Indeed, it would not have been possible to retrieve the value stored in the `TTarget` instance named `discriminant`, had we foregone the use of the temporary variable.

Our implementation of the root2.cc atomic class is not very robust since it does handle the special numerical cases whereby the polynomial coefficients `a` or `b` are zero, resulting in a division by zero at runtime. It would be a fairly simple task to extend the current implementation to make it more robust numerically. Consult the root2.cc atomic class provided as part of the global classes for a more robust implementation.

4.4.6 Basic Structure of a Multi-Valued EVALUATE Callback

The EVALUATE callback function of a multi-valued inverse implements a multi-dimensional function, also commonly referred to as a vector function, that calculates the values of the m target variables Y_j from the values of the n argument variables X_i. The target variables are the output values of the function, whereas the argument variables are the input values to the function.

$$\begin{cases} \mathbb{R}^n \to \mathbb{R}^m, (n,m) \in \mathbb{N}, m > 1 \\ F : X \mapsto Y = F\left(X\right), X \in \mathbb{R}^n, Y \in \mathbb{R}^m \end{cases} \tag{4.3}$$

The following code snippet shows the basic structure of the EVALUATE callback function defined for a multi-valued inverse calculating the values for (M+1) target ports from (N+1) argument ports. This code can serve as template for the callback function of your own multi-valued inverse, whereby the number of target and argument ports must be adapted and the code that calculates the result values must be added.

```
EVALUATE( callback_name )
{
// Declare (N+1) argument variables
    ARGUMENT( 0, arg_0 );
    …
    ARGUMENT( N, arg_N );

// Declare (M+1) target variables
    TARGET( 0, target_0 );
    …
    TARGET( M, target_M );

// Calculate (M+1) result values for all target variables
    double result_0 = …;
    …
    double result_M = …;

// Assign (M+1) result values to corresponding target variables
    target_0 = result_0 ;
    …
    target_M = result_M ;
}
```

5 MODELS OF PHYSICAL SYSTEMS

The previous examples were purely mathematical in nature. They allowed us to discuss the basic ideas in *SPARK*, unencumbered by details. Here we take up some of the other issues that arise when modeling physical systems. In particular, we show how *SPARK* handles the problems of unit consistency and range of values for variables. Also, we show provision in *SPARK* for modeling at a level higher than individual equations. Then, using these new ideas, we show the development of a *SPARK* model for a system of modest complexity.

5.1 UNITS, VALID RANGE, AND INITIAL VALUES

When simulating real physical systems, there must be consistency in the units of measure throughout the problem. In terms of a *SPARK* problem specification, this means that the units of a problem variable linked to an object port must be the same as the units assumed for the port variable when the object class was defined.

SPARK has a limited capability to ensure unit consistency. This is provided by associating an optional **unit string** with each port. Then the *SPARK* processor can check and report an error if you inadvertently connect variables of different units. Also, you can give initial, minimum, and maximum values for the port variable. For example, the **cpair** class from the HVAC Toolkit has a port for the specific heat coded as follows:

```
port CpAir “Specific heat of air” [J/(kg_dryAir*deg_C)]
           INIT = 1.0
           MIN = 0.01
           MAX = 5000.0;
```

The unit string is placed in square brackets `[…]`. Any connection to this port will have to have an identical unit string. The `MIN` and `MAX` values have the obvious meaning; run time warnings are issued when the value is outside this range. The `INIT` value is used by *SPARK* as the default starting value for the initial time solution if none is provided elsewhere. For example, if the associated variable happens to be a break variable, then the very first iteration will use the `INIT` value of 1.0 for **CpAir**.

In order for *SPARK* units checking to work to your benefit you must define a consistent set of units. Table 5-1 shows the SI units used in the HVAC Toolkit (see Appendix B). Other consistent sets could be used instead. Note that the units and value ranges given in are not built into *SPARK*; they are simply the units employed in the HVAC Toolkit class library. However, they do serve as an example of a consistent set of units. When developing *SPARK* models you have the choice of adhering to these units or developing your own library with units of your choice. Obviously, you should be consistent with whatever unit system you choose, otherwise you will have to implement special unit conversion objects when your objects are connected. The `INIT`, `MIN`, and `MAX` values should be set as appropriate for each port.

Table 5-1: SPARK Units (SI) used in the HVAC Toolkit

Unit String	Description	Initial	Minimum	Maximum
`[-]`	Unspecified			
`[J/kg_dryAir]`	Enthalpy, air	25194.2	-50300.0	398412.5
`[J/kg_water]`	Enthalpy, water	25194.2	-50300.0	398412.5
`[kg_water/kg_dryAir]`	Mass ratio	.002	0.0	0.1
`[kg_dryAir/s]`	Mass flow rate, air	10000.	1000.	1000000.
`[kg_water/s]`	Mass flow rate, water	10.	0.	1000.
`[deg_C]`	Dry-bulb temperature	20.	-50.	95.
`[m^3/kg]`	Specific volume, fluid		1.0	

[m^3/kg_dryAir]	Specific volume, air		0.6	1.6
[kg/m^3]	Ratio of total (air plus moisture) mass to volume	1.2026	0.6	1.8
[J/kg]	Enthalpy, steam			
[J/(kg*deg_C)]	Specific heat, fluid	1.0	0.01	5000.0
[J/(kg_dryAir*deg_C)]	Specific heat, air	1.0	0.01	5000.0
[kg/s]	Mass flow rate, fluid		0.0	
[m^3/s]	Volumetric flow rate, fluid		1	
[m]	Distance		1	
[m^2]	Surface area		0	
[W]	Power	1	-10000	10000
[Pa]	Pressure	101325	0	110000
[W/deg_C]	U*A, heat transfer	0	-1.0E6	1.0E6
[s]	Time, seconds	0.0	0	1.0E30
[fraction]	Any ratio	1.0	0.0	1.0
[scalar]	Any non-dimensional	1.0	-1.0E30	1.0E30

To demonstrate, consider the **sercond** class from the HVAC Toolkit, which models two conductors in series. The ports are defined as:

```
PORT U1   "Conductance 1"         [W/deg_C];
PORT U2   "Conductance 2"         [W/deg_C];
PORT Utot "Overall conductance"   [W/deg_C];
```

Then, when the **sercond** class is used in a problem definition you have to give matching unit strings at each `LINK` or `INPUT` statement for the problem variables connected to the ports of **sercond**:

```
DECLARE sercond  sc;
LINK    UA1 sc.U1         [W/deg_C]  INPUT  REPORT;
LINK    UA2 sc.U2         [W/deg_C]  INPUT  REPORT;
LINK    UATotal sc.UTot   [W/deg_C]         REPORT;
```

The *SPARK* parser can then check to be sure you have not made a units error; if the units string in a `LINK` or `INPUT` statement does not match those of all port variables in the same statement, a units error will be reported.

There are times when you may not want strict enforcement of unit consistency. For example, the **sum** class is used in many places, sometimes adding heat flux and other times mass flow rates. If we insisted on strict unit consistency, we would have to have a separate **sum** class for every different case. To avoid this problem, and to allow for problems where units are not important, there is an unspecified unit identifier. Units on a port are unspecified when you do not give any unit information, or when you explicitly declare unspecified units with "`[-]`" as the unit identifier. When a port has unspecified units, no unit checking is done on links to that port.

5.2 MACRO CLASSES

SPARK uses a computational graph based on individual problem variables and equations to produce an efficient solution strategy optimized for simulation speed. The *SPARK* atomic class is the fundamental building block where the equations are described. Because of this unique approach, *SPARK* is referred to as an equation-based solver.

While this is an advantage for efficient solving, the disadvantage is the tedium of defining a large system model entirely in terms of individual equations. When modeling physical systems, it is sometimes more convenient to work in terms of larger elements, such as models of physical components or subsystems. Such models most often will involve several equations and variables rather than one.

Macro classes allow you to work at a high level of abstraction, while allowing SPARK to employ efficient, equation-based solution strategies.

The macro class provides the abstraction mechanism for allowing more complex *SPARK* classes. It allows multiple atomic classes, and even other macro classes, to be assembled into a single entity for use by the model builder. Macro classes are used in problems or in other macro classes exactly like atomic classes, i.e., by use of the DECLARE keyword. However, when processed by the *SPARK* parser, any declared macro objects are separated into atomic objects so that the graph-theoretic solution methods can be applied in the normal manner.

As an example of the need for a macro class, consider the flow of air in a duct network, such as might occur in a heating system for a building. In simulation of these systems there is a need for models of various components such as diverters that split the flow into two streams and mixers that merge the flow of two duct sections into one. Here, let's focus on the mixer and devise a model for it in the form of a *SPARK* macro class.

The diagram in Figure 5-1 shows the mixer component.

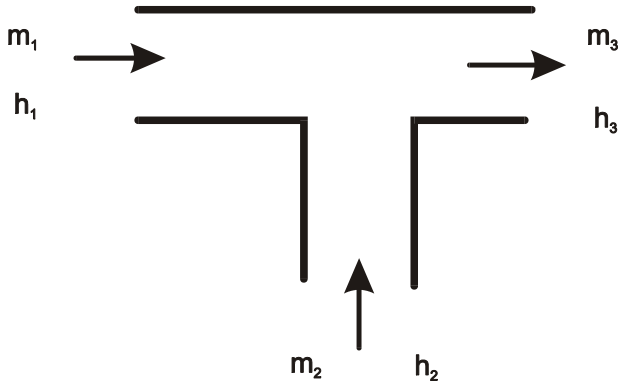

Figure 5-1: Dry air mixer.

The air duct mixer model must include two laws from physics: conservation of mass and conservation of energy. These can be expressed in the following equations:

$$\begin{aligned} m_1 + m_2 &= m_3 \\ m_1 h_1 + m_2 h_2 &= m_3 h_3 \end{aligned} \qquad (5.1)$$

where *m* represents mass flow rate and *h* represents the enthalpy of the air streams. The subscripts *1* and *2* represent the conditions at the two inlets, and *3* the condition at the outlet.

To construct a macro object class for the mixer let's assume that we already have object classes for the mass and energy balance equations. Actually, the mass equation can be represented with the familiar **sum** class. Also in the *SPARK* object library there is an object class called **balance** that represents equations like the enthalpy one. The port variables analogous to *m* and *h* are **m** and **q** respectively.

The macro class will connect the constituent classes exactly as if we were creating a problem definition file. Constituent class port variables that are to have the same meaning in the context of our new macro class are linked together, forcing equivalence. Those that are to be available for interfacing to problems or other macro classes are "elevated," i.e., made port variables of the macro class.

Figure 5-2 shows this idea and serves as a guide in writing the macro class. Because all represent the same quantity, the port variable **m1** of the macro class must be connected to the **a** port of the **sum** class and the **m1** port of the **balance** class. Other port variables are linked in a similar manner. The *SPARK* expression of this is shown below.

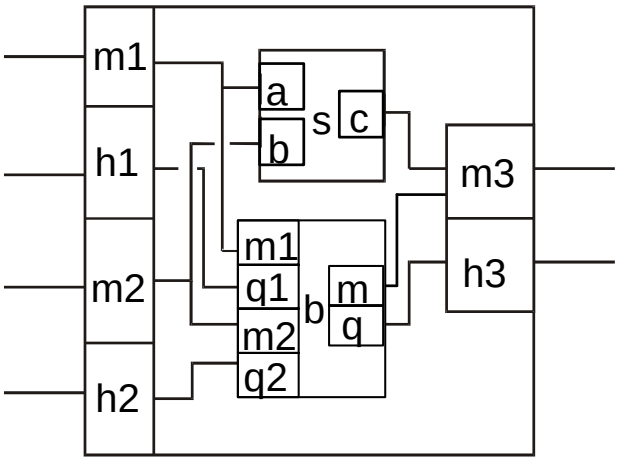

Figure 5-2: Mixer macro class diagram.

```
/* SPARK Mixer Object Macro Class
 *
 */
PORT  m1 "Stream 1 mass flow rate" [kg_dryAir/s];
PORT  m2 "Stream 2 mass flow rate" [kg_dryAir/s];
PORT  m3 "Stream 3 mass flow rate" [kg_dryAir/s];
PORT  h1 "Stream 1 enthalpy" [J/kg_dryAir];
PORT  h2 "Stream 2 enthalpy" [J/kg_dryAir];
PORT  h3 "Stream 3 enthalpy" [J/kg_dryAir];
DECLARE  sum  s;
DECLARE  balance b;
LINK  mass1 .m1, s.a, b.m1;
LINK  mass2 .m2, s.b, b.m2;
LINK  mass3 .m3, s.c, b.m;
LINK  enthalpy1  .h1, b.q1;
LINK  enthalpy2  .h2, b.q2;
LINK  enthalpy3  .h3, b.q;
```

It will be observed that this is very much like a problem definition. The principal difference is the absence of inputs. Also, note that a macro class has ports, whereas a problem does not. Ports provide the interface to the outside. That is, when an object of this class is used, connections will be made to its ports. The internal links, on the other hand, are not exposed to the outside at all. If you want a variable represented by a macro class link to be available for outside connections, you must connect it internally to a port. For example, the line:

```
   LINK  mass1 .m1, s.a, b.m1;
```

means that the link named *mass1* connects the **m1** port of the **mixer** macro class to the **a** port of *s* and the **m1** port of *b*.

Note the dot (**.**) in front of the first connection following the link names in the above example. The rationale for the dot syntax is based on the general connection notation **x.p**, where we are referring to the **p** port of the *x* object. When the port in question belongs to the macro class being defined, as opposed to one of its constituents, the class name is that of the very class we are defining and therefore is not expressed.[9]

The similarity between macro classes and problems makes it common practice when developing a macro class to first test it as a problem. For example, you could develop the mixer class as a problem, saving it in a file with .pr extension. Once it is working properly, you simply change the inputs to links, add ports for the variables needed at the interface, connect the corresponding links to these ports, and save it as a .cm file.

You may have noticed in the above example that the names of links, e.g., *mass1*, are not used anywhere. This is because we express the internal connections entirely in terms of the class and port names, as in **s.a**, or with an implied class name and port name as in **.m1**. Because link names are not used, they are optional when defining macro classes. That is, we could write:

```
LINK .m1, s.a, b.m1;
```

instead of the previous statement with exactly the same effect. In contrast, link names are required for problems, as these are the names by which we know the problem variables. Further discussion of link names is provided in Section 8.2.

Note that we have included unit strings in the ports. This will prevent you from connecting inappropriate links to objects of the **mixer** class. Also, we could have placed unit strings in the links to allow unit checking of the links to the ports of the classes which are used in the macro. We elect not to do so here, however, because both **sum** and **balance** are mathematical classes with generic ports.

Finally, note that macro classes are entirely equivalent to normal *SPARK* classes in terms of usage. They can be used in creating problem specification files or in building other macro classes. The *SPARK* parser recursively expands the macro objects as it generates the solver code.

[9] In some object-oriented languages, such as *C++*, the name of the class being defined is known internally as *this*. In SPARK we chose to have the name *this* be understood rather than expressed.

6 DIFFERENTIAL EQUATIONS

Thus far we have focused on problems with only algebraic equations. However, many simulation problems are dynamic in nature and involve differential equations as well. That is, some of the problem variables appear as derivatives with respect to time. In this Section we see that *SPARK* is capable of representing and solving such problems. We begin with a brief review of numerical methods used in solving ordinary differential equations.

6.1 NUMERICAL SOLUTION OF DIFFERENTIAL EQUATIONS

Numerical solution methods for differential equations start with given initial values for the dynamic variables[10] and attempt to project to a new solution a short time later. When the differential equation is part of a larger system of equations the entire set must be solved at each point to ensure accuracy. The process is then repeated, with the newly calculated values becoming the basis for the next projection forward. The amount by which time is advanced at each projection is called the **time step**. It is initialized with the value of the key `InitialTimeStep` specified in the run-control file. Generally speaking, the time step has to be small in order to achieve sufficient accuracy of the solution. Since simulations are often carried out over long periods of time, many small time steps are required. Computational efficiency is therefore very important.

The projection is done by means of an **integration formula** involving current and/or past values of the dynamicvariables and their time-derivatives. For example, the simple Euler integration formula *f* is:

$$x = f\left(x_p, \dot{x}_p\right) = x_p + h\dot{x}_p \tag{6.1}$$

where x is the dynamic variable, $\dot{x}$ is its derivative with respect to time, and *h* is the time step. Note that the Euler formula involves variable and time-derivative values only from the **previous** time, indicated by the subscript *p*. This is called an **explicit** formula because it gives the new solution explicitly, i.e., without reference to unknown values at the end of the current time step. On the other hand, some integration formulas do involve values of the dynamic variables at the new time, i.e.,

$$x = f(x_p, \dot{x}_p, x, \dot{x}) \tag{6.2}$$

Such formulas are called **implicit** because they involve values at the new point as well as past values.[11] Obviously, iteration might be required for implicit integration formulas, while not for explicit integration formulas. The aim of the more complex formulas is to get improved accuracy and numerical stability with larger time steps.

SPARK deals with differential equations by introducing object classes to represent integration formulas. These can be from the *SPARK* globalclass library, or user defined. You can define many different kinds of integration object classes, ranging from simple explicit formulas such as Euler's to complex implicit formulas used in predictor-corrector methods. Unlike other simulation languages, *SPARK* even allows you to use different integration formulas in different parts of the same problem.[12]

Below we will learn how to solve a simple differential equation. First we will use integrators from the *SPARK* library, and then see how integrator object classes are created. In Section 6.5, this will be extended to a more complex problem with mixed algebraic and differential equations.

[10] The dynamic variables are the variables appearing in differential form in the set of equations.

[11] The terms *open* and *closed* are sometimes used instead of explicit and implicit.

[12] This is however not recommended as there are numerous numerical issues involved with mixing integration schemes in the same problem.

6.2 SOLVING A SIMPLE DIFFERENTIAL EQUATION

As a simple example, consider the differential equation:

$$\dot{x} + ax = b; \qquad x(t_0) = x_0 \tag{6.3}$$

where $\dot{x}$ is understood to be the derivative of x with respect to time, *t*, the independent variable. We see this to be a well-posed problem: $x(t)$ can be determined given the parameters a and b, and the initial condition x_0.

To achieve a numerical solution in *SPARK* we view the derivative as a separate dependent variable. In order to preserve the balance between equations and variables, this additional variable requires an additional equation to be added to the set. An integration formula provides this needed equation, giving the value of x at the next point in time. If we employ the Euler formula, Equation (6.1), the set of equations to be solved is:

$$\begin{cases} \dot{x} + a \cdot x = b \\ x = x_p + h \cdot \dot{x}_p \end{cases} \tag{6.4}$$

It is seen that we again have a well-posed problem consisting of two equations in the two variables x and $\dot{x}$. Since both equations are algebraic, they can be easily solved by the established *SPARK* methodology.

This example is simple, but the method is general. Regardless of problem complexity, we simply introduce a new problem variable for every (first order) time-derivative, and at the same time introduce an integrator object for the dynamic variable.

The *SPARK* solver then has an algebraic problem to deal with. Observe also that implicit integration formulas, Equation (6.2), require no special consideration. Such formulas involve the x at the new time, i.e., are implicit in x. But this is of no concern, because the *SPARK* solver anticipates that an iterative solution process may be necessary due to the possibility of other cycles in the problem. The implicit integration formula is simply one more equation to be converged through the normal iteration.

One other issue needs to be dealt with, and that is preserving past values of dynamic variables and their derivatives. From Equation (6.1) we see that the Euler integration formula uses values of x and $\dot{x}$ from the previous time to calculate x at the new time. Some integration formulas use values of these quantities from earlier time steps as well. In order to provide these past values, *SPARK* provides four past values for all problem variables. This allows definition of a wide range of practical integrator classes.[13]

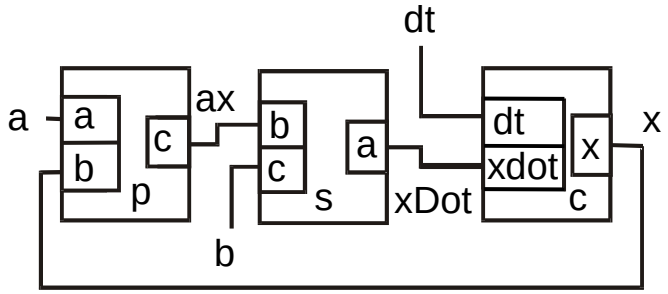

Figure 6-1: First-order differential equation diagram.

With these ideas we can continue with our example. Figure 6-1 shows a *SPARK* diagram for our differential equation. We use an instance of the **safprod** object class, *p*, to form the ax product, and an instance of the

[13] If needed, SPARK can be reconfigured to allow more past values.

sum object class, *s*, to form the sum $\dot{x} + ax$. We then link the **a** port of *s* and the **xdot** port of the instance *c* of the **euler** class using a problem variable called *xDot*. This causes the **x** port of *c* to carry the problem variable, *x*, which we also link to one of the multiplicand ports of the object *p*.

```
/*        First order differential  equation
 *                xdot + a*x = b
 *               frst_ord.pr
 */
DECLARE  safprod    p;
DECLARE  sum        s;
DECLARE  euler      c;
LINK     a     p.a         INPUT;
LINK     b     s.c         INPUT;
LINK     dt    c.dt              GLOBAL_TIME_STEP;
LINK     x     p.b, c.x          REPORT;
LINK     xDot  s.a, c.xdot;
LINK     ax    s.b, p.c;
```

The values of the variables *a* and *b*, must be placed in an input file, frst_ord.inp. Also, when you solve differential equations it is necessary to provide initial conditions for each dynamic variable. In *SPARK* there are two ways to accomplish this. One way is to place `INIT=VALUE` in the `LINK` statement for the variable. Alternatively, you can specify the initial values by giving the initial time and associated initial values for the dynamic variables in the input file. This is preferable if you want to carry out parametric runs with different initial conditions without changing the problem specification file. To demonstrate the latter method, suppose *a* and *b* are both 1.0, the initial time is 0, and *x* has an initial value of 0. Then frst_ord.inp should be:

```
  3     a     b     x
  0     1     1     0
```

Since *x* is a dynamic variable rather than specified as input, its value will be read from the input file **only at start-up** (see Section 7.1). Some numerical integration methods require values of dynamic variables and their derivatives at times earlier than the initial time. When needed, these values can be provided in the same manner, using time values earlier than the problem initial time (i.e., negative time if initial time is 0).

Note that the units of time are not defined in *SPARK*, so you are free to choose whatever time units you wish and develop your differential equations to reflect your choice. For example, if in the above differential equations x is measured in meters and $\dot{x}$ is to be in meters/second, the coefficient *a* must have units of reciprocal seconds and *b* must have units of meters/second.

The run-control file needed to run this problem, frst_ord.run, is:

```
(
InitialTime     ( 0.0 ())
FinalTime       ( 5.0 ())
InitialTimeStep ( 0.015625 ())
FirstReport     ( 0.0 ())
ReportCycle     ( 0.03125 ())
InputFiles      ( frst_ord.inp ())
OutputFile      ( frst_ord.out ())
)
```

We ask for the solution over a time range of 0 to 5 seconds, with a time step of 0.015625.[14] The link for *dt* includes the `GLOBAL_TIME_STEP` keyword. This propagates the time step specified in the run-control file to wherever it may be needed in the problem and macro classes. The requested output at every other time step is written to `frst_ord.out`. The results are plotted in Figure 6-2, generated by opening `frst_ord.out` with Microsoft *Excel*. Alternatively, you could use the free-use plotting program provided with *WinSPARK*, *wgnuplot*, see Section 14.

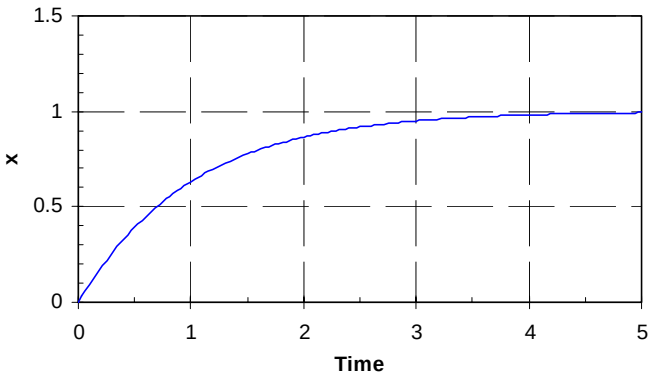

Figure 6-2: Results for the *frst_ord* dynamic problem.

6.3 INTEGRATOR CLASSES IN THE *SPARK* LIBRARY

The *SPARK* library has several integrator classes. All integrator classes have the same port interface consisting of the port **x** for the dynamic variable, the port **xdot** for its time-derivative, and the port **dt** for the global time step. In order to be able to distinguish the integrator objects from the other algebraic objects at runtime, each integrator class is also typed as `INTEGRATOR` using the `CLASSTYPE` statement (See Section 8.6).

This approach of typing the integrator classes and of keeping the same port interface for all of them allows to identify the dynamic variables and their respective time-derivatives in the problem under study. This information is then used during the solution phase in order to provide specific numerical treatment for the dynamic variables. In particular, classes of type `INTEGRATOR` are allowed to send requests to the solver to adapt the global simulation time step in order to satisfy the user-specified integration tolerance (See Section 10). Also, the list of dynamic variables is written out to the equation file generated by the *setupcpp* program (See Figure 1-1). The equation file should be consulted to find out which problem variables are the dynamic variables for which initial conditions must be provided at the start of the simulation.

If the `CLASSTYPE INTEGRATOR` statement is omitted in the atomic classes implementing the integration methods,[15] then the integrator objects are treated like any other atomic class[16] during the *SPARK* build process. The variables connected to the **x** port will not be tagged as dynamic variables and they will not be listed in the equation file. During the solution phase, these integrator objects will not be able to provide error control by adapting the time step. Notwithstanding these limitations, it is still possible to use integrator classes that are not defined as `CLASSTYPE INTEGRATOR`.

However, we strongly recommend that you use the new integrator classes of the *SPARK* library defined as `CLASSTYPE INTEGRATOR` in order to benefit from the improved numerical treatment such as the capability to monitor the integration error.

The integrator classes in the *SPARK* library are shown in Table 6-1. For each integrator, we also indicate whether it can be used with variable time step and whether it provides integration error control through varying the time step. All of the implemented integration methods are fully described in numerical analysis texts so we will just describe them briefly here.

[14] Although the time step can be any wanted value we choose $1/2^6$ =0.015625 because powers of 2 can be represented exactly in the binary storage format used internally. Step sizes that are not powers of 2 are difficult to synchronize with reporting intervals.

[15] This was the approach followed in *SPARK* 1.

[16] By default, the atomic classes are considered to be defined as `CLASSTYPE DEFAULT` unless specified otherwise.

Table 6-1: Integrator Object Classes in the *SPARK* Library.

Integration method	Class file	Variable Time Step	Error Control
Euler (explicit)	euler.cm	yes	no
Implicit Euler	implicit_euler.cc	yes	no
Backward-Forward Difference	bfd.cc	yes	no
4th-order Backward - Forward Difference	bd4.cc	no	no
Adams-Bashforth-Moulton	abm4.cc	no	no
PC Euler	integrator_euler.cc	yes	yes
PC Trapezoidal	integrator_trapezoidal.cc	yes	yes

- The Euler method is based on the simplest of all methods, using only the time-derivative at the beginning of the time step. It is a 1st-order integration method.
- The Implicit Euler method is the same basic idea as the normal (explicit) Euler method except the time-derivative is estimated at the end of the time step. It is also a 1st-order integration method but with better stability behavior than the explicit Euler method.
- The Backward-Forward Difference method is only slightly more complex, using the time-derivative at the end of the time step as well as at the beginning. It is a 2nd-order integration method.
- The 4th-order Backward-Forward Difference method uses additional previous values and time-derivatives. These Backward-Forward Difference methods are often used for "stiff" differential equations sets (Press, Flannery et al. 1988). This scheme cannot be used in the variable time step mode because the constant coefficient implementation assumes equidistant previous solution points.
- The Adams-Bashforth-Moulton method is a 4th-order predictor/corrector method. Such methods employ two separate integration formulas, a **predictor** to make an initial estimate of the new solution, and a **corrector** to refine the solution iteratively. The predictor is an explicit formula, while the corrector is an implicit formula. This scheme cannot be used in the variable time step mode because the constant coefficient implementation assumes equidistant previous solution points.
- The PC Euler method is a 1st-order predictor/corrector scheme providing control of the local truncation error when the solver is operated in the variable time step mode (See Section 18). The predictor scheme implements the explicit Euler method and the corrector scheme implements the implicit Euler method. The error estimate is obtained from the difference between the predictor and the corrector and is of order 1. The error control strategy uses the Euclidean norm of the local truncation errors estimated for each dynamic variable. The time step adaptive strategy implements the Error Per Step approach, whereby the error norm at each step is kept smaller than the user-specified tolerance.
- The PC Trapezoidal method is a 2nd-order predictor/corrector scheme providing control of the local truncation error when the solver is operated in the variable time step mode. The predictor scheme implements the explicit Euler method and the corrector scheme implements the trapezoidal method, also known as the backward-forward difference method. The error estimate is obtained from the difference between the predictor and the corrector and is of order 1. The error control strategy uses the Euclidean norm of the local truncation errors estimated for each dynamic variable. The time step adaptive strategy implements the Error Per Step approach, whereby the error norm at each step is kept smaller than the user-specified tolerance.

6.4 CREATING *SPARK* INTEGRATOR OBJECT CLASSES

6.4.1 Simplified Implementation of the Euler Method

If none of the library integrator object classes are suitable, you can define your own. *SPARK* integrator object classes are created much like any other object class. To see how this is done, let's look at the following implementation of the **euler** class. The port variables are the dynamic variable **x**, its derivative **xdot**, and the time step **dt**. An inverse is given for a single port variable, the dynamic variable **x**.[17]

```
/*          euler.cc              */
#ifdef spark_parser
PORT  x;
PORT  xdot;
PORT  dt;
EQUATIONS {
   x = x[1] + dt*xdot[1];
}
FUNCTIONS {
   x = euler__x(xdot, dt);
}
#endif  /*spark_parser*/
#include "spark.h"
EVALUATE( euler__x )
{
   ARGUMENT( 0, xdot);
   ARGUMENT( 1, dt);
   TARGET( 0, x);
   double result;
   result = x[1] + dt*xdot[1];
   RETURN( result );
}
```

The EVALUATE callback function `euler__x` employed in the class definition is implemented in *C++* after the class itself. It is basically an expression of the Euler integration formula, Equation (6.1). First, the arguments declared for the callback function `euler__x` are defined as instances of the class `TArgument` using the preprocessor macro ARGUMENT called with the respective positions in the argument list[18]: the time-derivative argument `xdot` and the time step argument `dt`. Then, the target for the dynamic variable `x` is defined as an instance of the class `TTarget` using the preprocessor macro TARGET.

The heart of the callback function is the line:

```
   result = x[1] + dt*xdot[1];
```

which represents the Euler formula. As might be surmised from the code, `x[1]` refers to the value of the target port **x** one time step back.[19] Similarly, `xdot[1]` refers to the value of the argument port **xdot** one time step back. The right hand side adds the time step multiplied by the derivative at the beginning of the time step to the variable at the same time. This is the new value of the dynamic variable, which is then returned using the preprocessor macro RETURN.

[17] Theoretically, *SPARK* would not care whether the integration formula was used to calculate the dynamic variable or its derivative. As a token to the sensibilities of most numerical analysts, however, here we restrict this relationship to be a formula for the dynamic variable.

[18] The ARGDEF macro uses zero-based indexing.

[19] Remember that it is not possible to access the value at the current step of a `TTarget` instance but the values at previous steps can be freely accessed as well as other properties.

Finally, note that we did not define the **euler** class as `CLASSTYPE INTEGRATOR`. However, this would be trivial to do by simply adding the statement in the class definition because our simplified implementation is already compatible with the port interface required by an `INTEGRATOR` class.

6.4.2 The Initialization Issue

The function `euler__x` in this example is simplified because there is no guarantee that the dynamic variable x will be equal to the prescribed initial value x_0 at the initial time t_0 in order to satisfy the prescribed initial condition[20]

$$x(t_0) = x_0 \tag{6.5}$$

To understand the initialization issue, we need to consider the situation at the very beginning of the simulation period, i.e., `InitialTime`, and contrast it with conditions at later time steps. At `InitialTime`, presumably we want the prescribed initial values of the dynamic variables to be used as shown in Equation (6.5). However, at all other times in the dynamic solution process we need to calculate the value of *x* from the integration formula used in the *SPARK* integrator object. That is, assuming we are using the Euler formula, we want to enforce:

$$x = x_p + h\dot{x}_p \tag{6.6}$$

where the subscript *p* refers to the previous time step values for *x* and its time-derivative $\dot{x}$.

Thus we see that the system model is slightly different at `InitialTime`. Ideally, then, we should formulate the problem twice, once with an object representing Equation (6.5) and again with the integrator relationship, Equation (6.6), starting the simulation with the first formulation and switching to the second after the `InitialTime` solution. However, *SPARK* cannot change the model during simulation; it allows for a single problem formulation. Therefore we have to use the integrator object at `InitialTime` as well as throughout the simulation period.

An approach to achieve proper start-up is to modify the integrator class to behave differently at `InitialTime`. For example, we could write

```
if ( ACTIVE_PROBLEM->IsStaticStep() )
   result = x.GetInit();
else
   result = x[1] + h*xdot[1];
```

where `ACTIVE_PROBLEM->IsStaticStep()` is a boolean function that returns `true` only when the problem under study is currently solving a static simulation step, which is typically the case when the time equals `InitialTime`. Also, `x.GetInit()` is a method invocation that returns the initial value of `x`. The `htm/chm` tutorial *SPARK Atomic Class API* should be consulted for more information on the `TTarget` and `TProblem` classes.

This is actually quite a good solution to the start-up problem. It is easy to implement and will adapt to even complex integrators. The drawbacks are small losses in computational efficiency and generality. The principal efficiency loss is due to the extra if-check which must be executed at every time step in the simulation; it is doubtful that this increase in solution time will be significant in most problems. The loss in generality is because certain kinds of initial conditions, e.g.,

$$\dot{x}(t_0) = c \tag{6.7}$$

[20] We also refer to the initial condition for a dynamic variable as its initial value.

cannot be enforced because they require solving a different initial problem. Future versions of *SPARK* will deal with this start-up situation more rigorously. Two different problem graphs will be constructed, one using a start-up formula and the other a proper integration formula. This will allow determination of completely different solution sequences at start up if needed to enforce special initial conditions. Moreover, this approach will permit use of different integration formulas whenever necessary later in the simulation, e.g., after a change in integration time step.

6.4.3 The Restart Issue

After initial time the solver might need to restart the simulation by computing a static step to calculate a consistent, new set of "initial" values for the dynamic variables at the current time. This operation is sometimes referred to as a warm restart[21] because it occurs after the initial time.

Resetting the integrators following a warm restart is a similar task to the one previously discussed with the initialization issue whereby the integration formula is bypassed and a constant value is returned for the dynamic variable. The difference with the warm restart is that this constant value is simply the value at the previous step.

For example, we could write

```
if ( ACTIVE_PROBLEM->IsStaticStep() )  // No integration
   result = (
      ACTIVE_PROBLEM->IsInitialTime() ?
      x.GetInit() // Initial time solution special case
      :
      x[1] // Use past value for restart after initial time solution
      );
else // Perform integration
   result = x[1] + h*xdot[1];
```

where `ACTIVE_PROBLEM->IsInitialTime()` is a boolean function that returns `true` only when the global time in the problem under study is equal to the initial time. Only at initial time do we return the initial value of the dynamic variable. For each static step after the initial time, we return the previous value of the dynamic variable, as expected for a warm restart.

6.4.4 The Previous Value Issue

More complex integrators, differing primarily in the use of more previous terms, may be found in the *SPARK* globalclass directory. There it will be seen that `x` two steps back is written `x[2]`, and so on. Users with special needs can reconfigure *SPARK* to work with any number of previous values of any class argument.

In addition to the initialization issue, the integrator in this example is also simplified in another way. As presented, it uses the variable name, *xdot*, that represents both the new value at the current time step and the previous value. However, the integration formula only needs knowing about its previous value. That is, the `euler__x` callback function has the form:

```
   x = euler__x(xdot, dt);
```

Written this way, the *SPARK* parser will assume that we are using the current-time value of *xdot* in the right hand side of the integration formula, whereas in fact it is the previous-time value of *xdot* that occurs there as can be seen in Equation (6.1). Since the code for the corresponding *C++* callback function `euler__x` actually uses only the previous value of *xdot* on the right hand side, namely `xdot[1]`, Euler integration will be properly applied at execution time.

[21] The initial time solution is also referred to as a cold start.

However, the disadvantage of the way we have coded it here is that the generated solver will potentially include an unnecessary feedback loop, and therefore an additional but unnecessary break variable, if the variable *xdot* depends on the variable *x* in the problem. A better way to implement explicit integrators that avoids this undesired topological dependency in the computational graph is discussed in Section 8.3.

6.5 SOLVING A LARGER EXAMPLE: THE AIR-CONDITIONED ROOM

As a more realistic simulation example, let us consider a simple air-conditioned room shown in Figure 6-3.[22]

The room is supplied by air at temperature T_{in}. The flow rate of supply air is $\dot{m}$, which is controlled by a proportional controller acting in response to the difference between room air temperature, T_a, and the set point, limited between maximum and minimum values $T_{\max}$ and $T_{\min}$. Heat Q_{wall} is transferred through the external envelope in proportion to the outside-to-inside temperature difference. Also, heat Q_{floor} is transferred from the floor slab to the room air in proportion to the temperature difference between these two bodies. Accounting for the heat capacity of the floor slab, the mathematical model for this system can be written:

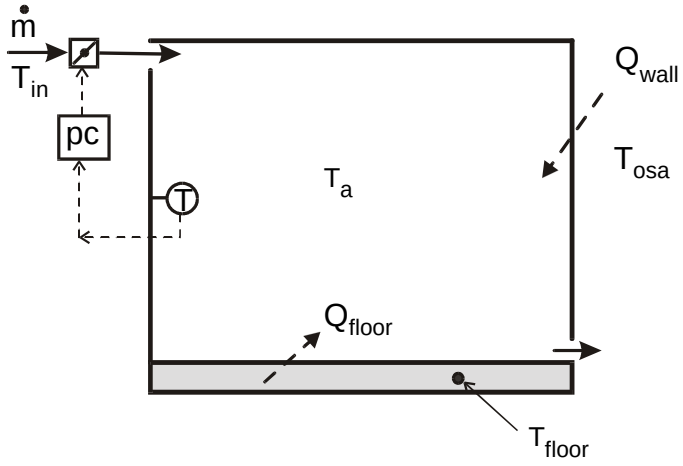

Figure 6-3: Temperature-controlled room.

$$
\begin{aligned}
Q_{wall} &= UA_{wall}\left(T_a - T_{osa}\right) \\
Q_{floor} &= hA_{floor}\left(T_a - T_{floor}\right) \\
Q_{flow} &= \dot{m}C_p\left(T_{in} - T_a\right) \\
Q_{floor} &= Q_{flow} - Q_{wall} \\
M_{floor}C_{p,floor}\dot{T}_{floor} &= Q_{floor} \\
\dot{m}C_{p,a} &= \begin{cases} M_{\min} & if\ T_a < T_{\min} \\ M_{\min} + \left(T_a - T_{\min}\right)\cdot\dfrac{M_{\max} - M_{\min}}{T_{\max} - T_{\min}} & \\ M_{\max} & if\ T_a > T_{\max} \end{cases}
\end{aligned}
\tag{6.8}
$$

where:

UA_{wall}	is the wall conductance,	Q_{flow}	is the heat added (+) or removed (-) from the room due to air flow,
T_{osa}	is the outside air temperature,	$\dot{m}C_p$	is the supply air capacity rate,

[22] The *VisualSPARK Users Guide* contains a tutorial showing how a similar problem would be formulated using the *VisualSPARK* user interface.

hA_{floor}	is the floor to room air conductance,	$M_{floor}C_{p,floor}$	is the floor slab heat capacity,
T_{floor}	is the floor slab temperature,	$M_{\max}$	is the maximum supply air capacity rate,
T_a	is the room air temperature,	$M_{\min}$	is the minimum supply air capacity rate,
Q_{wall}	is the heat flow from room air to walls,	$T_{\min}$	is the room temperature at which supply air capacity rate is maximum,
Q_{floor}	is the heat flow from room air to floor,	$T_{\max}$	is the room temperature at which supply air capacity rate is minimum.

The first two equations express the relationship between the temperature differences and heat flow to the room air, while the third gives the heat removal rate due to the stream of conditioned air. The next two give, respectively, the heat storage rate of the floor slab, Q_{floor}, and the rate of change of energy stored in the slab, $M_{floor}C_{p,floor}\dot{T}_{floor}$; of course, these quantities are equal.

The last equation is the proportional control expression, stating that the air stream cooling capacity is proportional to the difference between room air temperature and the set point, limited between maximum and minimum values.

This system can be represented by seven *SPARK* objects, as shown in Figure 6-4. The three heat transfer equations are represented by the objects *flow*, *walls*, and *floor*, all of which are instances of the HVAC Toolkit class called **cond** (a conductor) having the form:

$$q = U12 \cdot (T1 - T2) \tag{6.9}$$

The slab heat storage rate relationship is represented by a **diff** object called *net*. Also, a **safprod** object called *rate* is required to form a product between the slab heat capacity, *MCp*, and the rate of change of slab temperature, *T_floor_dot*. An integrator object called *c* implements the backward-forward difference formula to get *T_floor* from *T_floor_dot*. Finally, the proportional controller is implemented by the class called **propcont** from the HVAC Toolkit (see Appendix B).

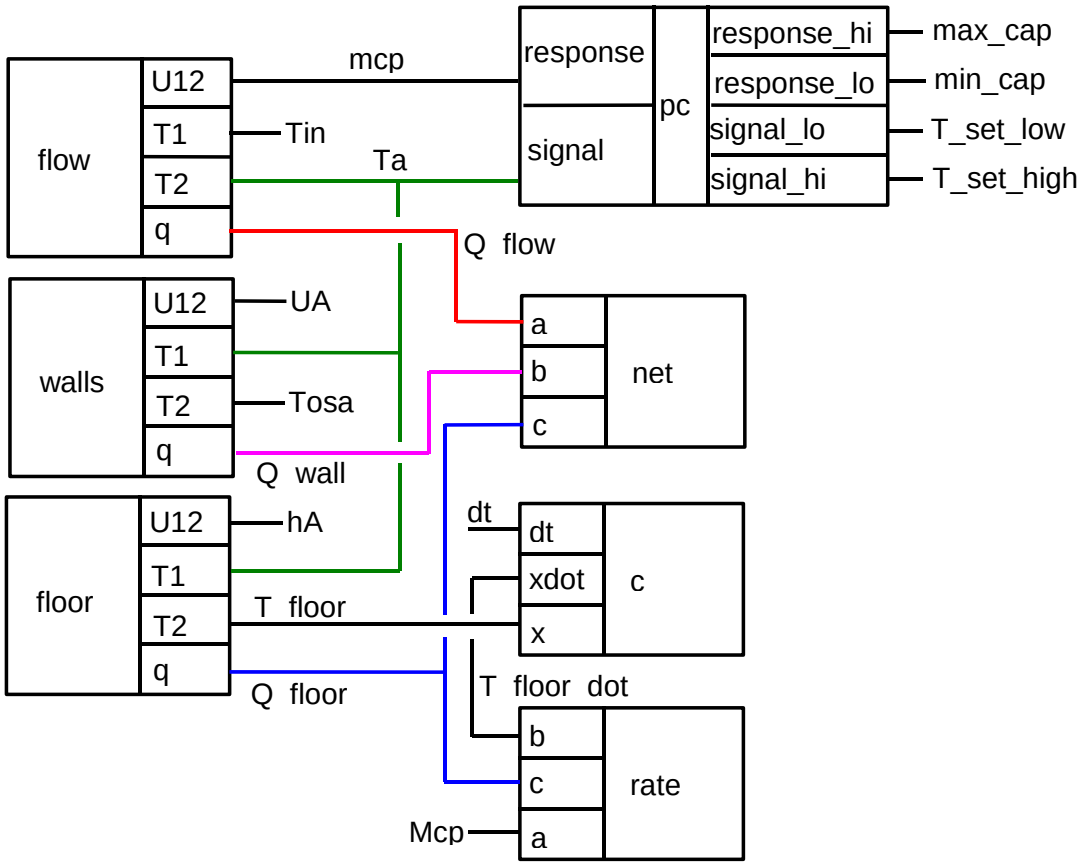

Figure 6-4: *SPARK* diagram for temperature-controlled room (see file room_fc.cm).

Because several rooms are often required in a complete problem, we implement the diagram in Figure 6-4 as a *SPARK* macro class called room_fc.cm, as shown below:

```
/*
     Massive Floor Room, with Controller Macro room_fc.cm
 */
// Temperatures
PORT  Ta          [deg_C]   "Room air temperature";
PORT  T_floor     [deg_C]   "Room floor temperature";
PORT  T_floor_dot [deg_C/s] "Room floor temperature rate of change";
PORT  Tosa        [deg_C]   "Outside air temperature";
PORT  Tin         [deg_C]   "Supply air temperature";

PORT  UA          [W/deg_C] "Wall conductance";
PORT  hA          [W/deg_C] "Floor to air conductance";
PORT  mcp         [W/deg_C] "Supply air heat capacity rate";
PORT  Mcp         [J/deg_C] "Floor mass heat capacity";

// Proportional controller
PORT  T_set_high  [deg_C]   "Set point temp, high";
PORT  T_set_low   [deg_C]   "Set point temp, low";
PORT  max_cap     [W/deg_C] "Max supply air capacity rate";
```

```
PORT  min_cap     [W/deg_C] "Min supply air capacity rate";

// Heat transfers
PORT  Q_flow      [W]       "Heat added (+) /removed (-) by air stream";
PORT  Q_wall      [W]       "Wall heat transfer";
PORT  Q_floor     [W]       "Heat from air to floor";

PORT  dt          [s]       "Time step for T_floor differential";

DECLARE  cond     flow;   //  Air mass flow "conductor"
DECLARE  cond     walls;  //  Walls conductance
DECLARE  cond     floor;  //  Floor to air conductor
DECLARE  diff     net;    //  Diff between Q in and out
DECLARE  safprod  rate;   //  Multiply T_floor_dot* Mcp
DECLARE  propcont pc;     //  Proportional controller
DECLARE  bfd      c;      //  Backward-forward difference integrator

LINK  .Tosa,        walls.T2;
LINK  .Tin,         flow.T1;
LINK  .UA,          walls.U12;
LINK  .hA,          floor.U12;
LINK  .mcp,         flow.U12, pc.response;
LINK  .Mcp,         rate.a;
LINK  .T_set_low,   pc.signal_lo;
LINK  .T_set_high,  pc.signal_hi;
LINK  .max_cap,     pc.response_hi;
LINK  .min_cap,     pc.response_lo;
LINK  .Q_wall,      walls.q,  net.b;
LINK  .T_floor,     floor.T2, c.x;
LINK  .T_floor_dot, rate.b,   c.xdot;
LINK  .Q_floor,     floor.q,  net.c,    rate.c;
LINK  .Ta,          flow.T2,  walls.T1, floor.T1, pc.signal   INIT=20.0;
LINK  .Q_flow,      flow.q,   net.a;
LINK  .dt,          c.dt;
```

This macro can be used to define a single-room problem as follows:

```
/*                Air-conditioned Room room_fc.pr */
DECLARE room_fc  room;

LINK  Mcp        room.Mcp         [J/deg_C]   INPUT;
LINK  UA         room.UA          [W/deg_c]   INPUT;
LINK  hA         room.hA          [W/deg_C]   INPUT;
LINK  Tosa       room.Tosa        [deg_C]     INPUT;
LINK  Tin        room.Tin         [deg_C]     INPUT;
LINK  T_set_high room.T_set_high  [deg_C]     INPUT;
LINK  T_set_low  room.T_set_low   [deg_C]     INPUT;
LINK  max_cap    room.max_cap     [W]         INPUT;
LINK  min_cap    room.min_cap     [W]         INPUT;

LINK dt          room.dt          [s]                  GLOBAL_TIME_STEP;
LINK mcp         room.mcp         [W/deg_C]                 REPORT;
LINK Q_flow      room.Q_flow      [W]                       REPORT;
LINK Q_wall      room.Q_wall      [W]                       REPORT;
LINK Q_floor     room.Q_floor     [W]                       REPORT;
LINK Ta          room.Ta          [deg_C] BREAK_LEVEL=10    REPORT;
```

```
LINK T_floor     room.T_floor     [deg_C] INIT=30          REPORT;
LINK T_floor_dot room.T_floor_dot [deg_C/s]                REPORT;
```

Here we have declared *room* as an instance of the **room_fc** macro class. The room thermal characteristics and control settings are defined as inputs. This alone would be sufficient to completely specify the problem since the necessary linkages are all internal to the **room_fc** macro class. However, if we did not put some `LINK` statements in the problem file, *SPARK* would have no problem variables and hence nothing to report. We therefore introduce `LINK` statements to get reports on the room air temperature, *Ta*, floor slab temperature, *T_floor*, cooling rate of the air stream, *Q_flow*, and the air stream capacity rate, *mcp*. Alternatively, you could use the `PROBE` keyword (see Section 8.5).

The input data for this problem is shown in Table 6-2. Note that the supply air temperature is initially 13°C, and is raised to 17°C at 20 hours (72,000 seconds) after starting. The `room_fc.inp` file to specify this is constructed as shown below:

```
9       hA   UA   Tosa   Tin   Mcp    T_set_low   T_set_high   max_cap   min_cap
0       60   30   38     13    1.e6   23          24           50        0
71964   60   30   38     13    1.e6   23          24           50        0
72000   60   30   38     17    1.e6   23          24           50        0
*
```

In the first line the first item, 9, is the number of problem input variables. The next nine items in this line are the names of the input variables as defined in the `INPUT` statements in the problem specification file. The data that follow give the times (in this case, seconds) and values for the inputs at discrete points throughout the intended simulation period. The first line, with a time value of 0, gives the initial conditions. We specify *Tin* to be set at 13°C from time 0 to 19.99 hours (71,964 seconds), and 17°C from 20.0 hours (72,000 seconds) forward. Other values are constant throughout the simulation. *SPARK* will interpolate linearly between the given time values to arrive at the value of all input variables at each solution point as the simulation proceeds.[23] The last line has an asterisk, *, meaning that all values remain fixed from that point forward.

It will be observed that the time unit in the above example is seconds. While there is a certain awkwardness with this choice, it has the advantage of allowing the other problem variables to be expressed in true SI units. For example, had we chosen to use hours instead of seconds, the time values would be the (perhaps) more pleasing sequence 0, 19.99, 20.00, but then we would have had to express input data such as *hA* in J/(hour*deg_C) instead of W/deg_C.

Another observation in this example is that some input values do not vary with time, and this leads to many repeated values in the file. While there is nothing wrong with repeating the constant values as done here, there are alternatives that you may want to consider. Perhaps the best way to deal with this situation is with multiple input files, as discussed in Section 7.6.2. Another way to deal with a constant input variable, not necessarily recommended, is simply to omit it from the input file. This sometimes works because problem input variables not listed in an input file will assume their `INIT` values, if available. `INIT` values are specified in the `PORT` statement (Section 19.10) when *SPARK* classes are defined. If the class does not provide `INIT` values, or the provided values are not acceptable, you can also give an `INIT` value on a link connected to the port. The disadvantage of doing it this way is that the problem must be rebuilt whenever `INIT` values are changed.

However provided, running the *room_fc* problem with the data in Table 6-2 produces the results plotted in

[23] Note that there must be some time difference between successive points to allow legitimate interpolation.

Figure 6-5 and Figure 6-6.[24]

Table 6-2: Input for the Temperature-Controlled Room Example.

Variable **see Equation** (6.8)	**Link** **see** room_fc.pr	**Value**	**Units**
hA_{floor}	hA	60	W/deg_C
UA_{wall}	UA	30	W/deg_C
T_{osa}	Tosa	38	deg_C
$M_{\min}$	min_cap	0.0	W/deg_C
$M_{\max}$	max_cap	50	W/deg_C
$T_{\min}$	T_set_low	23	deg_C
$T_{\max}$	T_set_high	24	deg_C
dt	dt	360	s
$M_{floor}C_{p,floor}$	Mcp	1.0E6	J/deg_C
$T_{in}(0-71964)$	Tin	13	deg_C
$T_{in}(72000-\ldots)$	Tin	17	deg_C
$T_{floor}(0)$	T_floor	30	deg_C

All inputs are constant except *Tin*, which starts at 13°C and is increased to 17°C at 20 hours (72,000 s). The first of these plots,

Figure 6-5, shows the controlled quantity, *mcp*, and we see that it remains at its maximum value for about six hours. During this period the room air temperature, Figure 6-6, is being rapidly reduced. Once within the range of proportional control, the supply capacity rate modulates, maintaining the room air temperature close to the set point. The slab temperature gradually cools. At the twentieth hour, the scheduled change in supply air temperature takes place, causing the supply capacity rate to increase to the maximum. However, this maximum is insufficient so the air temperature rises above the set point.

[24] To get these plots we opened the output file with Microsoft Excel. Alternatively, *gnuplot* could be used.

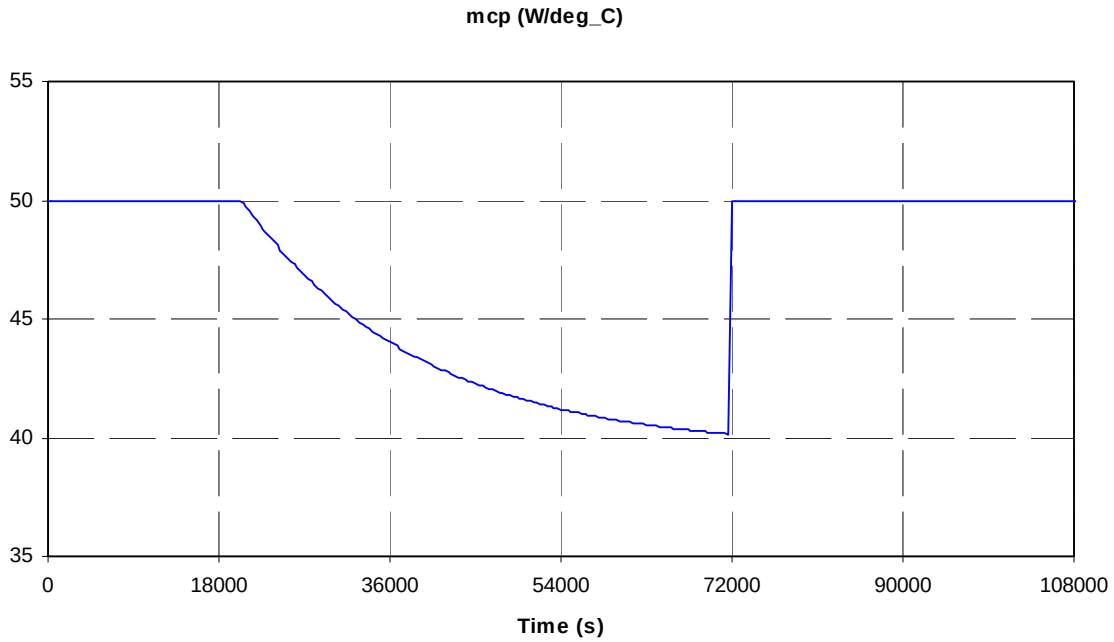

Figure 6-5: Supply Air Capacity Rate.

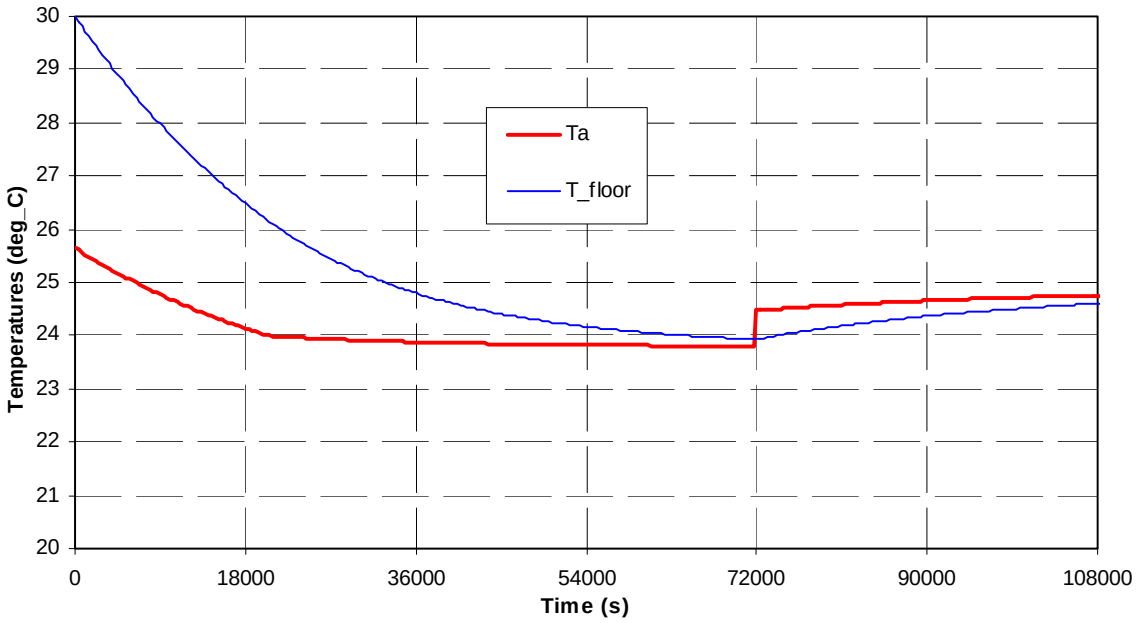

Figure 6-6: Room and Floor Slab Temperatures.

The room_fc.eqs file, shown below, reveals how *SPARK* solves this problem. We see that there is a single strongly connected component, with one break variable, *Ta*. The initial value of *Ta* is taken from the `INIT` values found in the macro or underlying atomic classes, since it is not mentioned in the input file, and no `INIT` value is given in the `LINK` statement in the problem file.

The dynamic variable *T_floor* is initialized at `InitialTime` with the `INIT` value given in the `LINK` statement in the problem file. After the initial time it is computed by the integrator object `room`c` using its time-derivative *T_floor_dot* and the global time step *dt*.

These, along with the problem inputs, allow the indicated sequence of calculations. The component is iterated to convergence at each time step.

```
Global variable(s) :
        GLOBAL_TIME_STEP                = dt


Known variable(s) :
        max_cap                         INPUT
        min_cap                         INPUT
        T_set_high                      INPUT
        T_set_low                       INPUT
        Mcp                             INPUT
        hA                              INPUT
        UA                              INPUT
        Tosa                            INPUT
        Tin                             INPUT


Dynamic variable(s) :
        T_floor                         <- room`c( T_floor_dot, dt )


Component 0 :
   Break variable(s) :
        Ta                              PREDICT_FROM_LINK = Ta

   Solution sequence :
        mcp                             = room`pc:propcont__response( Ta, T_set_low,
T_set_high, min_cap, max_cap )
        Q_flow                          = room`flow:cond_q( Tin, Ta, mcp )
        Q_wall                          = room`walls:cond_q( Ta, Tosa, UA )
        Q_floor                         = room`net:diff__difference( Q_flow, Q_wall )
        T_floor_dot                     = room`rate:safprod__a_or_b( Q_floor, Mcp )
        T_floor                         = room`c:bfd__x( T_floor_dot, dt )
[BREAK] Ta                              = room`floor:cond_T1( Q_floor, T_floor, hA )
```

7 HOW *SPARK* ASSIGNS VALUES TO VARIABLES

In a broad sense, one would think that variable values in a problem should either be user-specified or calculated in the process of solving the problem. While this is indeed true, there are issues having to do with *SPARK* value assignments that sometimes need careful attention. This is best discussed in terms of four different methods of value assignment that can take place in *SPARK*: initialization, prediction, updating, and solution.

7.1 INITIALIZATION

Initialization refers to providing values that are needed at the beginning of the simulation. Using these initial values, *SPARK* then computes values for all link variables at the initial time of the simulation. While all *SPARK* variables can be initialized, not all need to be initialized.

7.1.1 What Must be Initialized

There are two cases where variables must be given initial values, regardless of the numerical methods to be used:

Dynamic variables. These are the link variables that appear in differential equations, i.e., those attached to an **x** port of integrators. This initialization requirement arises directly from the underlying mathematical theory, namely that you need an initial condition, in addition to the differential equation, in order to have a well-posed problem. This requirement is independent of the choice of integration method or other numerical considerations.

Previous-Value Variables. Previous-Value Variables (see Section 8.3) are in a special category in *SPARK*. Most *SPARK* non-input link variables get values in the process of solving the problem equations at the time in question. Previous-Value Variables, on the other hand, get their values from calculations done **at the previous time step**.
As described in Section 19.14, the syntax `INPUT_FROM_LINK=fromLink` defines the link from which the variable in question gets its value. For this to work properly at `InitialTime`, obviously the variable referred to as `fromLink` must be initialized at the time **one time step before the problem initial time**. This can be done either in an input file, or using the `INIT` in the `LINK` statement defining `fromLink`. Note, however, that Previous-Value Variables that arise in the definition of integrators need not be initialized because they are not used at `InitialTime`. (See Section 8.3)

7.1.2 What Might Need Initialization

Additionally, certain numerical integration methods may need to be initialized not only at `InitialTime`, but also at one or more earlier time steps. While this can be done in *SPARK*, as a practical matter it is difficult or impossible to know such values.[25] Ideally you should attempt to provide past values as needed by multistep methods, if used. That said, some analysts may be willing to accept some degree of inaccuracy in early time steps, in which case this advice can be disregarded.

BDF-like multistep schemes require past values for the dynamic variables, as many as the order of the method. For example, the **bd4** class requires values at one, two, three, and four time steps before the initial time of the simulation. Similarly, Adams-like multistep schemes, e.g., the classes **bfd** and **abm4** require past values for the derivatives of the dynamic variables, again as many as the order of the method.

[25] For this reason, *SPARK* will avoid use of such methods at the beginning of the simulation and until necessary histories of past values have been solved for with single step methods.

Finally, it should be noted that the variables that *SPARK* selects as break variables may need initialization. The reason for this is that unless the `LINK` statement for break variable has the keyword `PREDICT_FROM_LINK=fromLink` (see below) the iteration process at each new time begins at the previous value of the break variable. Without proper initialization, the previous value at `InitialTime` would likely become the built-in *SPARK* default value, 0.01. To override use of the default value, you must initialize the break variable at `InitialTime`.

7.1.3 How to Specify Initialization

You can specify initial values in two ways. First, `INIT=value` can be placed in the `LINK` statement for the variable, or in any equivalent link to a `PORT` statement in macro objects (see Sections 8.1and 19.14). An alternative way to initialize is by means of input files. During the initialization phase of the simulation, all variables can have initial and past values assigned through reading from input files. This is done by providing the required variables and derivatives with values at `InitialTime`, and earlier time steps if needed, using negative times if necessary.

Initialization of Previous-Value Variables is a special situation. Since a variable of this kind gets its value from the previous value of another variable, the proper way to provide its `InitialTime` value is to specify the value of the corresponding `fromLink` at one time step before `InitialTime`, indicated by initial time minus the time step, using time stamp prior to `InitialTime` in an input file. Note that an attempt to use the `INIT` keyword in a `LINK` statement in which the `INPUT_FROM_LINK` keyword is used results in a warning. Moreover, values given for Previous-Value Variables per se in input files will be ignored.

Thus we see that *SPARK* initial values can come from the default values, `INIT=value`, or input files. Either of the latter two will override the first. If a variable has both `INIT=value` and occurs in an input file, the file input overrides the `INIT` value.

7.1.4 Initial time solution of a dynamic problem

To ensure starting the integration process from a consistent set of initial values at the first dynamic time step after the initial time, *SPARK* computes the initial time solution of a dynamic problem by solving a surrogate static problem (derived from the dynamic problem description), whereby all integrator objects return the initial value of the dynamic variable instead of evaluating the corresponding integration formula (See Section 6.4.2). Thus, the dynamic variables will be set to their desired initial values and the algebraic variables will be solved to produce a consistent solution at the initial time.

The following steps are carried out by the initial value loader in *SPARK* prior to solving the problem at initial time:

1. Load the `INIT` property values for all problem variables from the hard-coded values specified in the *.pr, *.cm and *.cc files. The default `INIT` value, if none has been specified explicitly in the problem specification, is 0.01.
2. Read in the `INIT` property values from input files specified for the initial time stamp. Note that only the `INIT` properties of the variables appearing in the input files are updated. The other variables keep their `INIT` values as specified in the *.pr, *.cm and *.cc files.
3. Propagate the `INIT` values to the current values of all problem variables.
4. Read in the current values from input files for all problem variables specified for the initial time stamp, thus overwriting the previously loaded `INIT` values. This ensures that values specified in a snapshot file used to restart the simulation are loaded correctly, overriding prior `INIT` specifications.
5. Propagate the initial values to the `INPUT_FROM_LINK` variables, if any.

6. Finally, write the current values back to the `INIT` property for each problem variable, so that subsequent firing of the method `TArgument::GetInit()` returns the correct initial value resulting from the previous steps. This last step ensures proper initialization at `InitialTime` of the dynamic variables through the usage of the `TArgument::GetInit()` method in the integrator classes (See Section 6.4).

7.2 PREDICTION

In *SPARK* prediction refers to providing values for break variables at the beginning of each time step, i.e. prior to solving the simultaneous algebraic problem by iteration.

7.2.1 Where Prediction is Needed

As a rule, only break variables need predicted values.

7.2.2 How Prediction is Specified

By default, predicted values for break variables come from the final value for the same variable found at the previous time step. In many cases this will work well, so you don't have to take any special steps. If your problem encounters solution difficulties, you may want to provide better prediction using either the `PREDICT_FROM_LINK` feature for links, or the `PREDICT` feature in the class definition.

If `PREDICT_FROM_LINK=fromLink` appears in the `LINK` statement for a break variable, the starting value for the iterative solution at the new time will be the value of `fromLink`. This mechanism is used when you know that the value of `fromLink` provides a more reliable estimate for the break variable than its previous value. Note that since the `fromLink` can be any link, this mechanism allows you to devise predictor using variables from anywhere in your problem. Therefore it is a very general and powerful mechanism.

Another mechanism for prediction is provided by the syntax:

```
PREDICT = predictor_fun(port1, port2, port3, ...)
```

in the `FUNCTIONS` segment of a *SPARK* class definition. This methods provides a predictor at the **class level**, as opposed to the `PREDICT_FROM_LINK` keyword which provides prediction at the **link level**. Class-level prediction is primarily used to implement predictor-corrector integration schemes (e.g., `abm4.cc`), where the predictor scheme is specified following the `PREDICT` keyword. Another possible usage of class-level prediction is to provide a predictor function for a nonlinear atomic class using a linearized form of the nonlinear equation. This approach has been successfully applied with the airflow-pressure power law relation in the zonal model context. Unlike link-level predictors, class-level predictors can involve only the variables connected to the ports of the class in question.

If a variable has both link and class level prediction (an unlikely situation), the class level prediction will override the link level prediction.

7.3 UPDATING

The concept of Previous-Value Variables (see Section 8.3), requires the concept of updating as a means of assignment of values to such variables.

7.3.1 What Needs to Be Updated

Updating refers only to providing values for Previous-Value Variables at the beginning of each time step.

7.3.2 How Updating is Specified

To implement this concept, every Previous-Value Variable has in its defining `LINK` statement:

```
INPUT_FROM_LINK = fromLink
```

Previous-Value Variables are viewed as receiving values by updating from the specified links. At the beginning of every time step, before solving the problem equations, the saved previous value of `fromLink` is assigned to the variable named in the `LINK` statement.

7.4 SOLUTION

Solution is the prevalent method whereby values are assigned to variables in a *SPARK* problem.

7.4.1 What Needs to Be Solved For

Normally, values for *SPARK* variables are determined by the solution of the system equations at each time point in the solution interval. The exceptions to this are, input variables, previous-value variables, and dynamic variables at `InitialTime.`

7.4.2 How Solution Is Specified

As noted earlier, keywords in the associated `LINK` statements often determine the role of the variable. Inputs variables are identified by the keyword `INPUT` either replacing the `LINK` keyword, or occurring elsewhere in the `LINK` statement. Previous-Value Variables are defined by the keyword `INPUT_FROM_LINK` in the `LINK` statement. Dynamic variables, on the other hand, have no special identifying keyword. Variables become dynamic merely by being connected to an **x** port of an integrator. The absence of these special keywords in a `LINK` statement indicates that the associated variable is to be solved for.

Break variables are normal *SPARK* variables, other than inputs or Previous-Value Variables, that happen to be selected by *SPARK* for iteration. Although they are assigned predicted values at the beginning of iteration at each time step, their final values after convergence at each time step are “solution” values, i.e., they satisfy the system equations. Note that the break variables are determined automatically by *SPARK*.

7.5 PROPAGATION

As discussed previously, *SPARK* problem variables can have a default value assigned through the use of keywords in the `PORT` statement. This default value will replace the built-in default value (0.01) for the port. However, when *SPARK* atomic classes are used to build macro classes, and when both become parts of *SPARK* problem files, a question arises about precedence among these values as set at different levels.

For example, suppose we define atomic class **ac1** which has a port called **T** with a default value of 20. Now suppose we define a macro class **mc1** which uses **ac1**, and this class also has a port called **T** with a default value of 10 which is linked to the **T** port of **ac1**. The question is, which default value will *SPARK* use for variables linked to the **T** port of the class **mc1** when it is used in a problem or another macro class? The same question can be posed for the `INIT`, `ATOL`, `MIN`, and `MAX` values assigned through the `PORT` or `LINK` statements.

These questions are answered by propagation rules built into the *SPARK* parser. The first rule is that the **higher level takes precedence**. This means that a `DEFAULT`, `INIT`, `ATOL`, `MIN`, and `MAX` values given at any level override those given in lower level ports to which there is a connecting path. That is, values will automatically propagate downward as needed. Thus if **mc1** were to be used in a problem file (or another macro class), any variable linked to its **T** port would have a default value of 10.

Let us consider another facet of this problem. Suppose a default value is not given for the **T** port of the **mc1** class discussed above. Will a variable linked to the **T** port of the **mc1** class have a default value (other than the built-in value of 0.01) when it is used in a problem or another macro class? The rule given above addresses downward propagation, but this question is one of upward flow of information, from a port in a low level class to a port linked to it in the higher level class or problem. To deal with this situation, *SPARK* applies a second propagation rule, which is that `DEFAULT`, `INIT`, `ATOL`, `MIN`, and `MAX` values are propagated upward through connected ports whenever the higher level ports have no corresponding values.

Together, these propagation rules produce behavior that most users will find natural. However, ambiguity can arise when a macro class port is linked to two or more ports of constituent classes. For example, suppose **mc1** also uses another atomic class, **ac2**, which also has a `PORT` called **T**, but with a default value of 15. Will the value propagated upward (in the absence of default specification of the **T** port in **mc1**) be 20 or 15? There is no way for *SPARK* to resolve such an ambiguity. Consequently, the propagated value will be determined by the order in which the parser encounters the linkages in **mc1**. To avoid such ambiguity, you should assign values at the higher levels when building complex macro classes.

7.6 INPUT VALUES FROM FILES

Most *SPARK* problems require data beyond that which is specified in the problem specification file. In particular, as we saw in the examples of Section 2, variables designated as `INPUT` in the problem specification file need run time values. Moreover, certain other kinds of data are needed to specify exactly how the problem is to be solved numerically, e.g., initial values for dynamic variables and prediction values for iteration variables. All such data can be provided in *SPARK* input files. Although usually bearing the .inp extension, files of any extension can be used as *SPARK* input files.

7.6.1 Categorization of Different Types of Input

Although in simple examples we have dealt with in this manual so far we have used a single input file for a *SPARK* problem, in practice it is often better to segregate the different kinds of input into separate files. One useful categorization of different types of input is:

Constant data: These are usually physical characteristics of the system that do not change with time. For example, surface areas, equipment capacities, and any other physical problem data that are assumed to be constant, such as heat transfer coefficients.

Time-varying data: This includes any problem input data that varies with time during the simulation interval. The most common example in HVAC problems is weather data, such as ambient temperature and humidity. However, system control information, such as thermostatic set points, that are scheduled to change at particular times are also time-varying inputs.

Initial Conditions: If the problem includes differential equations, the initial values of all dynamic variables must be provided. Although these can be specified in the problem specification file with the `INIT` keyword, it is usually better practice to specify them in an input file so they can be changed in subsequent runs without rebuilding the problem.

Numerical support data: Numerical techniques used in *SPARK* sometimes need, or at least benefit from, additional user supplied data. This category often includes initial predicted values for variables that are solved for by iteration, i.e., break variables. Also, if the chosen numerical integration methods for differential equations in the problem require previous values of the dynamic variables and/or their derivatives, they belong in this category.

In a well organized problem, each of these categories should have a separate input file. Moreover, it is sometimes wise to have multiple files within these categories. For example, you could have a separate constant data file for each subsystem in a complex model. Another situation calling for multiple input files within a category is when time-varying data has different temporal characteristics. For example, if we wanted

to have outside temperature *Tosa* varying hourly in the *room_fc* problem example it would be far easier to place this in a different file than the one with *Tin* which changes only once.

7.6.2 Example of Multiple Input Files

We can demonstrate these ideas by revisiting the *room_fc* problem example from Section 6.5 For example, we could create four separate input files using the above categories. The constant data file, appropriately called room_fcDesignParameters.inp, would contain:

```
 8   hA   UA   Tosa   Mcp    T_set_low   T_set_high   max_cap   min_cap
 0   60   30   38     1.e6   23          24           50        0
```

while the time-varying data file, that we might call room_fcTimeVaryingParameters.inp, would contain:

```
 1       Tin
 0       13
 71964   13
 72000   17
 *
```

Since the controlled room problem includes a differential equation, it is necessary to specify the initial value of the dynamic variable, *T_floor*. Rather than relying upon the `INIT` keyword to set the initial value for this dynamic variable we can specify it in an initial conditions input file. This file could be called room_fcInitialConditions.inp and would contain:

```
1   T_floor
0   30
```

One advantage of this approach is that it is not necessary to rebuild the problem when initial values change.

Finally, we should create an input file for whatever information is needed to support the numerical solution process, provided such information is available. One issue in this regard is initial predictions for break variables, as explained in Section 7.2. As explained there, at the very beginning of the solution an initial predictor is needed because otherwise there would be no “previous time value” to use. If a reasonable estimate for a break variable is not readily available, *SPARK* can sometimes find a solution beginning with the default initial value, 0.01. However, if you can estimate more appropriate initial predictions the iteration process will have a better chance of quickly finding the correct solution at the start of the problem. Note that while better accuracy of these initial predictors will improve the chances for solution, usually great accuracy is not necessary.

In the case of the controlled room example the equation file reveals that *SPARK* chooses *Ta* as break variables. For the *Ta* variable, we can easily provide an estimate more accurate than the default value. For example, a value half way between the initial *T_floor* value and the supply air temperature value should be a reasonable for *Ta*. Thus a numerical support input file called room_fcNumericalSupport.inp could therefore be created as:

```
1   Ta
0   21.5
```

A problem run-control file (see Section 18) must list the names and locations of all input files. For this example, we have room_fc.run as:

```
(
InitialTime     ( 0.0 ())
```

```
FinalTime        ( 108000.0 ())
InitialTimeStep  ( 180 ())
FirstReport      ( 0.0 ())
ReportCycle      ( 360.0 ())
InputFiles       ( room_fcDesignParameters.inp ()
                   room_fcTimeVaryingParameters.inp ()
                   room_fcInitialConditions.inp ()
                   room_fcNumericalSupport.inp ()
                 )
OutputFile       ( room_fc.out ())
)
```

8 ADVANCED LANGUAGE TOPICS

8.1 MACRO LINKS

When systems with fluid flow are modeled, the component models are often connected with a common set of links. For example, HVAC system air components such as fans, heating and cooling coils, and mixing boxes are connected by links representing air enthalpy (or temperature), humidity, and mass flow rate.

In *SPARK*, a set of ordinary links such as these can be grouped together and used as a **macro link**, connecting **macro ports** of classes, thereby simplifying specification of such models.[26]

As an example of macro links and ports, consider a moist air mixer in which we define the interface to have three macro ports, representing two inlet flow streams and one outlet flow stream:

```
PORT AirEnt1  "Inlet air stream 1"  [airflow]
   , .m    "air mass flow"       [kg_dryAir/s]
   , .w    "hum. ratio"          [kg_water/kg_dryAir]
   , .h    "enthalpy"  NOERR     [J/kg_dryAir]
   ;
PORT AirEnt2  "Inlet air stream 2"  [airflow]
   , .m    "air mass flow"       [kg_dryAir/s]
   , .w    "hum. ratio"          [kg_water/kg_dryAir]
   , .h    "enthalpy"  NOERR     [J/kg_dryAir]
   ;
PORT AirLvg   "Leaving air stream"  [airflow]
   , .m    "air mass flow"       [kg_dryAir/s]
   , .w    "hum. ratio"          [kg_water/kg_dryAir]
   , .h    "enthalpy"  NOERR     [J/kg_dryAir]
   ;
```

In this example, each macro port has three properties or subports, namely mass flow rate, humidity ratio, and enthalpy. Although the individual subports of one of these ports have separate names, description strings, and physical units, the macro port itself also has a name, description, and units string.[27]

When an object of this class is instantiated you can connect similar macro ports (i.e., those with like units and similar internal structure) in the same manner as you would connect ordinary ports. Thus if the class with the above interface were called **mixerMP** we could write (in some macro class or problem we were creating):

```
DECLARE mixerMP m1, m2;
LINK    AirStream1 m1.AirLvg, m2.AirEnt1;
```

This would connect the humidity ratio, mass flow rate, and enthalpy of the air stream leaving *m1* with the first inlet of *m2*.

Developing classes that use macro ports requires great care, since if it is not done correctly the objects will not connect properly. The principal requirement is that if the macro ports of two objects are to connect properly, the ports must be similarly defined in both objects. By "similarly defined," we mean that the unit strings for both macro ports must be identical, and that there must be at least one common port name between the two ports. This is no problem in the above example, since *m1* and *m2* are of the same class, and the leaving air port is defined exactly the same as the two entering ports.

[26] Technically, a macro link does not exist in its own right as a SPARK construct. It is just a term for referring to a link connected to a *macro port*.

[27] Although, rather than physical units, the macro port "units"" are merely a unique name, selected by the user.

However, errors can easily occur if the two ports being connected belong to objects of differing class, perhaps developed by different people. For example, suppose a fan class were to be defined with the entering air port defined as:

```
PORT AirEnt     "Inlet air stream"    [airflow]
   , .massFlow  "air mass flow"     [kg_dryAir/s]
   , .w    "hum. ratio"             [kg_water/kg_dryAir]
   , .h    "enthalpy"  NOERR        [J/kg_dryAir]
   ;
```

Since the units string , `airflow`, is the same, *SPARK* would allow the following connection to be attempted:

```
DECLARE mixerMP m1;
DECLARE mfan f1;
LINK    InFlow m1.AirLvg, f1.AirEnt;
```

However, since the **flow** subport is called **m** in the **mixerMP** class and **massFlow** in the **mfan** class, only the **w** and **h** subports would be successfully connected. This is because when the *SPARK* parser expands the macro link/port, it attempts to match subports of like names. If there are no subports in the second object that match any of the subports of the first, the parser rejects the `LINK` statement as erroneous. But if at least one of the subports at one end matches a subport at the other end, *SPARK* assumes you know what you are doing and accepts the link. This is useful since you may indeed want to connect some but not all subports; for example, you may wish to connect one component with a dry-air macro port (i.e., no humidity ratio) with another component that was designed for moist air calculations.[28]

There are also situations where you need to qualify an individual subport in a macro link with one or more keywords. For example, suppose the first inlet port of *m1* in our first example comes from problem input data, and the mass flow rate is to be reported. The syntax to accomplish this is shown below:

```
DECLARE mixerMP m1, m2;
LINK    AirStream1 m1.AirLvg, m2.AirEnt1;
INPUT   massFlow1  m1.AirEnt1.m REPORT;
INPUT   hFlow1   m1.AirEnt1.h;
INPUT   wFlow1   m1.AirEnt1.w;
```

As is seen in this example, this syntax is much the same as for ordinary links or inputs; the only difference is that we qualify the port name, e.g., **m**, with the subport name as a prefix. The dot (**.**) is used as a separator.

While the above syntax is valid and easy to interpret, it is not concise. A more concise syntax that expresses the same connections is:

```
DECLARE mixerMP m1, m2;
LINK    AirStream1 m1.AirEnt1 (.h) INPUT (.w) INPUT (.m) {INPUT REPORT};
LINK    AirStream2 m1.AirLvg, m2.AirEnt1;
```

The first `LINK` statement defines a macro link called *AirStream1* that is connected to the **AirEnt1** macro port of the *m1* object. We see that each subport is referenced with the notation (**.portName**), and that following such reference there is a keyword such as `INPUT` that applies only to that subport. If more than one keyword is needed, they are enclosed in braces, e.g., `{INPUT REPORT}`. Thus we see that all three subports are to come from input, and the **m** subport is to be reported.

The need to make direct subport connections also arises in defining classes that have subports. For example, the **mixerMP** class might be (partially) implemented using the concise syntax as:

```
DECLARE  enthalpy  e1, e2, e3;
DECLARE  sum       s;
DECLARE  balance   hb, wb;
```

[28] This is somewhat like plugging a 2-wire appliance cord into a 3-wire wall outlet.

```
LINK AirEnt1  .airEnt1,
        (.TDb) e1.TDb
        (.w) {e1.w, wb.q1}
        (.h) {e1.h, hb.q1}
        (.m) {s.a, hb.m1, wb.m1};
LINK AirEnt2  .airEnt2,
        (.TDb) e2.TDb
        (.w) {e2.w, wb.q2}
        (.h) {e2.h, hb.q2}
        (.m) {s.b, hb.m2, wb.m2};
LINK AirLvg   .airLvg,
        (.TDb) e3.TDb
        (.w) {e3.w, wb.q}
        (.h) {e3.h, hb.q}
        (.m) {s.c, hb.m, wb.m};
```

Here we see that each subport of the three macro ports is linked to the appropriate ports of the constituent enthalpy and balance objects. The normal syntax could also be used here, but this would require four times as many statements.[29]

8.2 INTERNAL *SPARK* NAMES FOR VARIABLES (FULL NAMES OF LINKS OR PORTS)

In our early examples the name of a problem variable was synonymous with the user-defined name assigned in a `LINK` or `INPUT` statement. For example, in:

```
DECLARE room r;
LINK    Ta r.Ta;
```

Ta is the link name and it obviously represents the variable placed at the **Ta** port of the *r* object, probably a room air temperature. However, due to the hierarchical nature of *SPARK* programming, there are places where internal names used by *SPARK* might not be quite so obvious. This matter can be important when you are reading certain *SPARK* files, such as the `.eqs` file for complex problems, and when using the `PROBE` keyword (see Section 8.5).

To understand *SPARK* naming conventions you must understand that at solution time the solver works entirely at the equation level. This means that when *SPARK* parses a problem file, all macro objects and macro links must be expanded into atomic objects and links. When this happens, link names in higher level objects are propagated downward, as might be expected, overriding names that may have been assigned in the class definition of lower level object. For example, suppose that the **room** class used in the above link statement is (partially) defined as:

```
DECLARE  cond      flow; /*  Air mass flow "conductor"    */
DECLARE  cond      walls;/*  Walls conductance            */
DECLARE  cond      floor;/*  Floor to air conductor       */
DECLARE  diff      net;  /*  Diff between Q in and out    */
DECLARE  propcont  pc;   /*  Proportional controller      */
LINK     Tair .Ta, flow.T2, walls.T1, floor.T1, pc.signal [deg_C];
```

From this we can see that the problem level link named *Ta* is known as *Tair* inside the room class, and is connected to the **Ta** port of that class, and to ports of various names of the constituent classes of room. By the noted propagation rule, all of these lower level names are overridden by the problem level name *Ta*.

[29] The **mixerMP** class is one of the many classes in the HVAC Tool Kit implemented in the macro port form.

As a result of this downward propagation of link names, all problem level variables are readily identifiable when reported, for example, in the .eqs file.

However, often there are links in lower level objects that do not appear at the problem level. This occurs whenever a macro class developer elects not to connect an internal link to a port, or if the user of the class elects not to connect some unessential port (i.e., one with the NOERR keyword. See Section 19.10). As an example, the **mixer** class in the HVAC Toolkit class library is defined as:

```
PORT  m  "Combined flow rate, e.g., total mass flow" ;
PORT  q  "Combined transported quantity, e.g., enthalpy" ;
PORT  m1 "First inlet flow rate" ;
PORT  q1 "First inlet transported quantity" ;
PORT  m2 "Second inlet flow rate" ;
PORT  q2 "Second inlet transported quantity" ;
DECLARE   safprod  sp1, sp2, sp;
DECLARE   sum      s;
LINK   .m, sp.a ;
LINK   .q, sp.b ;
LINK    c  sp.c, s.c ;
LINK   .m1, sp1.a ;
LINK   .q1, sp1.b ;
LINK    a   sp1.c, s.a ;
LINK   .m2, sp2.a ;
LINK   .q2, sp2.b ;
LINK    b  sp2.c, s.b ;
```

Note that the links named *a*, *b*, and *c* are not connected to ports. Consequently, they cannot be accessed from higher level objects, and therefore cannot be problem level variables.[30] Nonetheless, these links represent variables whose values must be calculated by the *SPARK* solver at run time, and they will be assigned names by the *SPARK* parser. Under normal circumstances, you would not need to know these names; after all, they are merely intermediate variables needed to solve the mixing equations. However, if your problem does not solve properly you may have to look in the .eqs file, in which case you may want to know the names *SPARK* assigns to such links. Also, if you need to use the PROBE keyword, you will need to know how to refer to lower level links and ports (see Section 8.5).

Link names that do not resolve to problem-level links are generated by concatenation of object, link, and port names beginning at the highest level at which the link appears and going down to the port of an atomic class. The special prefix symbols (single quote (`), tilde (~), and dot (.)) are used in the concatenation to ensure unambiguous names. As an example, if we declare a **room** in a problem file as:

```
DECLARE room r;
```

and the room declares a **mixer**:

```
DECLARE mixer mix1;
```

then the *c* link in the **mixer** would be referred to as:

```
r`mix1~c
```

This might be read “the *c* link in the *mix1* object in the *r* object.” The single quote (`) prefixes an object in a hierarchy of objects, while the tilde (~) prefixes links. In a more complex situation, objects may be nested deeper, for example,

```
obj1`obj2`obj3~linkname
```

[30] Unless the *probe* statement is used (Section 3.8).

Also, as mentioned in Section 5.2, links within a macro class are often unnamed. In this case, *SPARK* will use a generated string of the form "NONAMEn" where n is an integer. Thus you might see:

```
obj1`obj2`obj3~NONAME7
```

in *SPARK* equation files.

An additional complication is introduced when macro links are used (see Section 8.1). Since macro links may have several subports, the link name must be qualified with the name of the particular port of interest. For example,

```
obj1`obj2`obj3~linkname.p1
```

refers to the **p1** port of link *linkname* in *obj3* that is part of *obj2* that is part of *obj1*. And if the **p1** port itself was in fact a macro port, we could go on with:

```
obj1`obj2`obj3~linkname.p1.a
```

to refer to the **a** subport of the **p1** port of the link *linkname* in *obj3* which is part of *obj2* which is part of *obj1*. Fortunately, since you are primarily concerned with higher level problem variables, you don't often have to cope with this complexity.

8.3 PREVIOUS-VALUE VARIABLES, OR UPDATING VARIABLES FROM LINKS

As discussed in Section 7.4, most *SPARK* variables are determined by solution of the problem equations at the current simulation time. This means that each variable gets assigned a value that is calculated from an inverse of one of the problem equations. There are situations, however, when a variable in a simulation must represent the previous value of some other variable. Such a variable needs no equation since its value is determined merely by assignment of the value of some variable at the previous point in time. A variable of this nature can be called a **Previous-Value Variable**.

Since *SPARK* variables are carried on links, Previous-Value Variables are viewed as receiving values by inputting from specified links. Consequently, *SPARK* provides INPUT_FROM_LINK[31] as an optional keyword in a LINK statement, taking the form:

```
LINK linkName <connections> INPUT_FROM_LINK = FromLinkName;
```

At the beginning of the time step, before solving the problem equations, the saved previous value of FromLinkName is assigned to linkName. As discussed in Section 7.1 initializing a Previous-Value Variable must come from the INIT= keyword in the FromLinkName, not in the Previous-Value Variable link itself. Indeed, it is an error to place the INIT keyword in a LINK statement that contains the INPUT_FROM_LINK keyword. Alternatively, the initial value can come from .inp files as discussed in Section 7.1.

As an example we shall revisit the Euler integration formula discussed in Section 6.4. For simplicity there we implemented the Euler integration formula as a *SPARK* atomic class with a port **xdot** representing the time-derivative of the dynamic variable, and the name of this port was used in the argument list of the euler__x callback function, i.e.,

```
x = euler__x(xdot, dt);
```

However, as explained in Section 6.4.4 this results in unnecessary iteration since the *SPARK* parser will not know that, internal to the function, only the past value of *xdot* is used. We can use the INPUT_FROM_LINK

[31] The keyword INPUT_FROM_LINK had been named UPDATE_FROM_LINK in previous version of *SPARK* up to 1.0.1 The name was changed to INPUT_FROM_LINK to better reflect the behavior.

keyword to correct this deficiency as follows. First, we rename the atomic class as euler_formula.cc and the callback function as `euler_formula__x`.

```
/*          euler_formula.cc             */
#ifdef  spark_parser

PORT  x;        // Dynamic variable
PORT  xdot;     // Previous-value variable updated with INPUT_FROM_LINK
PORT  dt;       // Time increment

EQUATIONS {
   x = x[1] + dt*xdot[1] ;
   bad_inverses = xdot, dt ;
}

FUNCTIONS {
   x = euler_formula__x( xdot, dt);
}

#endif  // spark_parser
#include "spark.h"

EVALUATE( euler_formula__x )
{
   ARGUMENT( 0, xdot ) ;
   ARGUMENT( 1, dt ) ;
   TARGET( 0, x ) ;
   double result;

   if ( ACTIVE_PROBLEM->IsStaticStep() ) {  // No integration
      result = (
        ACTIVE_PROBLEM->IsInitialTime() ?
        x.GetInit() // Initial time solution special case
        :
        x[1] // Past value for restart after initial time solution
      );
   }
   else {  // Perform the actual integration
      result = x[1] + dt*xdot[1];
   }
   x = result ;
}
```

Note that we renamed this atomic class **euler_formula** in order to be able to define a new macro class called **euler** which conceals the complexity of the `INPUT_FROM_LINK` considerations and preserves the convenient interface used in the Section 6.2 examples. Here is the **euler** macro class:

```
/*          euler.cm             */
PORT x;
PORT xdot;
PORT dt;

DECLARE euler_formula e;
LINK  DT     .dt    e.dt;
LINK  X      .x     e.x;
LINK  XDOT   .xdot;
LINK  XDOT_p        e.xdot  INPUT_FROM_LINK = XDOT;
```

Internal to the macro class we create links for both the current and previous values of $\dot{x}$. The previous-value variable *XDOT_p* is updated from the current-time variable *XDOT*.

Note that a link name, not a port name, must follow the `INPUT_FROM_LINK` keyword. Due to this requirement we define the links *XDOT*, connect it the port **xdot**, and use it as argument to the `INPUT_FROM_LINK` keyword.

Finally, note that it is not necessary to initialize the previous-value variable in this example because, as a consequence of the if-statement in the function definition, they are not used at `InitialTime`, but only at the solution points after `InitialTime`.

There are uses for previous-value variables other than in integrators for solution of differential equations. For example, simulation of discrete time controllers requires past values, both to calculate controller "integral action" and to determine when to update the controller output. An additional usage is for introduction of an artificial time delay in a troublesome iterative loop. By simply making some variable in the loop a previous-value variable the need for iterative solution is removed. If the time step is short, the error introduced may be acceptable.

8.4 USAGE OF THE LIKE KEYWORD IN PORT STATEMENTS

`PORT` statement can have the `LIKE` keyword to copy the properties of another port, that was previously defined. The subports are also copied. The usage of the `LIKE` keyword has the form:

```
   LIKE = anotherPortName
```

Note that any other input that is specified in the current port statement overrides the copied information. In the example below the port statements using the `LIKE` keyword:

```
PORT AirEnt1 "Inlet air stream 1"    [airflow]
                      .m "air mass flow"        [kg_dryAir/s] MIN=0.1
                    , .w "hum. ratio"           [kg_water/kg_dryAir]
                    , .h "enthalpy" NOERR       [J/kg_dryAir] ;

PORT AirOutWithT "Outlet air stream" [airflowWithT]
                       LIKE=AirEnt
                      .T "air temp"             [deg_C]
                    , .m  MIN=3.4 ;
```

produce the same specifications for port **AirOutWithT** as:

```
PORT AirOutWithT "Outlet air stream" [airflowWithT]
           .m "air mass flow"        [kg_dryAir/s] MIN=3.4
         , .w "hum. ratio"           [kg_water/kg_dryAir]
         , .h "enthalpy" NOERR       [J/kg_dryAir]
         , .T "air temp"             [deg_C] ;
```

Here, when defining the port **AirOutWithT**, the subports of the port **AirEnt1** are copied, the `MIN=0.1` attribute of the subport **.m** is changed to `MIN=3.4`, and the new subport **.T** is added.

8.5 THE PROBE STATEMENT

As noted in the section 8.2, there are often *SPARK* links that are not visible at the next higher level due to not having been elevated to a port of the class in which they are defined. Yet, sometimes it is convenient or necessary to be able to gain access to such links from higher levels. For example, you may want to report the *c* link internal to the **mixer** class in Section 8.2. While you could solve this problem by editing the **mixer** class, i.e., adding a new port for *c*, this is not a good solution. First, making changes to widely used classes is

hazardous; errors might be introduced, or you might cause unwanted behavior in other applications that use it. Another reason to avoid this approach is that if the needed access is several levels up in a hierarchy, you will have to edit every class in the hierarchy to elevate the needed link to where it is needed. The PROBE statement is provided to give an easier and better solution to such problems. It allows you to reach down into lower level objects, either to report values or set DEFAULT, INIT, ATOL, MIN, or MAX values. You can also set MATCH_LEVEL and BREAK_LEVEL for the link.

The PROBE statement has the same general format as the LINK statement. However, you must use the full *SPARK*-generated name for the low level link, as explained in Section 8.2. As an example, we will use PROBE to set the INIT value and request reporting for the **c** port of the **mixer** class in the **room** class mentioned in Section 8.2:

```
PROBE mixer_c  r`mix1~c INIT=0.5 REPORT;
```

This statement would be put in the problem file in which the room *r* is declared. Here *mixer_c* is a user-defined name for the probe. The expanded name of the wanted lower level link is *r`mix1~c*. With the INIT keyword we set the initial value, to be used if this link was selected as a break variable for iterative solution, to 0.5. Finally, the REPORT keyword causes the value of the *c* link in the **mixer** class to be reported along with other requested report variables during solution. The probe name *mixer_c* will be used as the label in the requested reporting.

As an aside, it is interesting to note that the above statement could also be written as:

```
PROBE mixer_c  r`mix1`sp.c INIT=0.5 REPORT;
```

or as:

```
PROBE mixer_c  r`mix1`s.c INIT=0.5 REPORT;
```

In these alternative forms, we set the probe to point at the **c** ports of either the *sp* or *s* objects to which the *c* link is connected. Since the values on the ports will be the same as the value on the link at run time, the same values will be reported.

8.6 USAGE OF THE CLASSTYPE KEYWORD IN ATOMIC CLASSES

In *SPARK*, atomic classes are typed. An atomic class can be:

- an integrator class,
- a sink class, or
- a default class.

The class type is specified using the CLASSTYPE statement in the atomic class. If no CLASSTYPE statement is specified, *parser* assumes that the class type is DEFAULT.

```
CLASSTYPE [SINK | INTEGRATOR | DEFAULT];
```

Typing the atomic classes allows to provide special processing for these classes during the graph analysis and/or the numerical solution phase at runtime.

8.6.1 INTEGRATOR classes

INTEGRATOR classes implement an integrator object whereby a dynamic variable[32] connected to the **x** port is integrated using its time-derivative variable connected to the **xdot** port and the global time step connected to the port **dt**. A dynamic variable can only be connected to one INTEGRATOR object to ensure well-posedness of the DAE[33] system. Also, the names of the ports are fixed and cannot be changed for any INTEGRATOR class or it will generate a parsing error.

Usually, an INTEGRATOR class also provides a specific behavior for the initial time solution that consists in solving a static problem to ensure a consistent initial calculation. Typically, the EVALUATE callback returns the initial condition for the dynamic variable instead of computing the solution of the integration scheme (See Section 6.4).

Since all INTEGRATOR classes have the same port interface and define a unique inverse that is assigned to the port **x**, they all can be represented with the same directed graph shown in Figure 8-1.

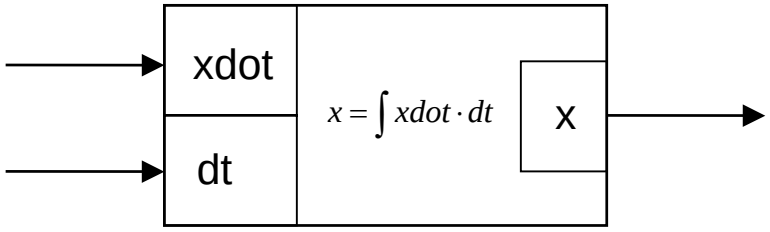

Figure 8-1: Directed graph representing an INTEGRATOR object.

8.6.2 SINK classes

A SINK class does not calculate any values. It acts as a sink node in the directed, computational graph from which no edge leaves. It allows you to define classes that do not directly participate in the calculation process. Therefore they are invoked at the very end of the solution sequence. Also, a SINK class can implement neither the EVALUATE nor the PREDICT callbacks (see Section 9).

The FUNCTIONS {…} statement for a sink class looks a bit different than for the other class types because there are intrinsically no target ports and there can be only one inverse per class, called the sink inverse. Furthermore, all the ports defined at the interface the atomic class must be listed as arguments of the callbacks comprising the inverse of the SINK atomic class. If some ports do not appear in any argument lists of the callback functions, then *parser* will generate an error. Finally, the unique inverse of a SINK class is named after the class name since we cannot use the name of the EVALUATE callback as with the other class types (See Section 3.1.3).

This following code snippet shows the class definition of a SINK class that only defines the CONSTRUCT and DESTRUCT callbacks.

```
CLASSTYPE SINK;

PORT x;

FUNCTIONS {
```

[32] Also called a differential variable. The other problem variables are referred to as algebraic variables.

[33] DAE stands for Differential-Algebraic Equation.

```
   CONSTRUCT  = fn_construct( x )
   DESTRUCT   = fn_destruct( x )
   ;
}
```

This class definition results in the directed graph without outgoing edge shown in Figure 8-2. `SINK` objects are treated as directed objects by the *setupcpp* program during the matching phase.

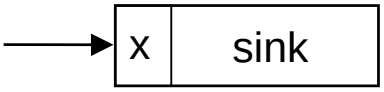

Figure 8-2: Directed graph representing a `SINK` object defining only the port **x**.

8.6.3 DEFAULT classes

The role of the default classes is to calculate the values for the target ports defined with each inverse. In particular, any atomic class from *SPARK* 1 falls into this category as well as the classes defined in the HVAC ToolKit library.

8.7 USAGE OF THE RESIDUAL KEYWORD IN EVALUATE CALLBACKS

8.7.1 Motivation

For complex equations, some inverses may be difficult or impossible to obtain as functions in **explicit form**.[34] Or, it may be that special knowledge about the problem under investigation suggests that a particular inverse should not be used, because, for example, it might lead to numerical difficulties such as:

- a division by zero resulting in an infinite number, or
- an invalid domain for a mathematical function (e.g., square root of a negative number), or
- an infinite partial derivative that would make the resulting Jacobian matrix badly conditioned.

Another situation where it is not desired to express an inverse in explicit form occurs when the inverse acts as a wrapper around a third-party program that calculates residual equations. Such a program, typically a legacy code, cannot easily be changed to return the values of the target ports. Instead you can embed it unchanged in a residual inverse. Note that it is also possible to define multi-valued inverses in residual form.

To deal with such situations, it is possible to specify inverses that do not return the values of the target ports but instead return the residual values for the equations assigned to each target port. Such an inverse is said to be expressed in **residual form**.

Clearly, this affects the way the `EVALUATE` callback defined for the residual inverse has to be implemented. Also, defining a residual inverse forces its target ports to be break variables because they must also appear as argument ports, therefore creating a de-facto algebraic loop in the resulting computational graph generated by the *setupcpp* program.

[34] This is achieved by symbolically rearranging the terms in the equation in order to produce an functional form that solves for the target variable, i.e. the target variables appear on the left-hand sign of the = sign. The terms defined on the right-hand side of the = sign then correspond to the function in explicit form for this target variable.

8.7.2 Implications for the Graph-Theoretic Analysis

For the graph-theoretic processing, a residual inverse is defined with a default match level of 4, as opposed to 5 for the other inverses in explicit form. This makes it less likely for the inverses in residual form to be chosen during the matching process. The rationale behind this design decision is that usually residual inverses are numerically less efficient to solve than their counterpart in explicit form because they tend to increase the resulting number of break variables.

Therefore, *setupcpp* favors using inverses in explicit form to inverses in residual form whenever possible. It is of course possible to overload this default match level in any `LINK` statement connected to the port assigned to the residual inverse (See Section 12.2).

8.7.3 Mathematical Example

As an example of a residual inverse we use the square_robust.cc atomic class that is part of the global classes found in the globalclass subdirectory.

```
/*+++
  Identification:  Square root of a value using a residual form to avoid
                   numerical problems due to badly-conditionned jacobian matrix
                   for small values of the square port.
---*/
#ifdef SPARK_PARSER


PORT root   "square^0.5" ;
PORT square "root^2"    MIN = 0 ;

EQUATIONS {
    square = root * root ;
}

FUNCTIONS {
   root    = RESIDUAL square_robust__root( root, square )  ;
   square  = square_robust__square( root ) ;
}

#endif /* SPARK_PARSER */
#include "spark.h"

// Residual form that is numerically more stable than the direct
// inverse (see square_root() in square.cc) for values of
// the port square close to zero.
//
// Also, the sign of the port square depends on the sign
// of square in the following way :
//
//           sign(root) = sign(square)
//
// Make sure that this convention is respected.
//
EVALUATE( square_robust__root )
{
   ARGDEF(0, root);
   ARGDEF(1, square);

   double residual = SPARK::sign(root)*pow(root, 2.0) - square;

   RETURN( residual );
}
```

```
// Same direct inverse as square_square() in square.cc
EVALUATE( square_robust__square )
{
    ARGDEF(0, root) ;
    double square  ;

    square = SPARK::sign(root) * root * root;

    RETURN( square );
}
```

The **square_robust** class calculates the square root of the value of the port **square** from the value of the port **root**. When the square value is negative the class models the symmetric function for the square root of the absolute value. Figure 8-3 shows the graphical representation of the equation modeled by the class.

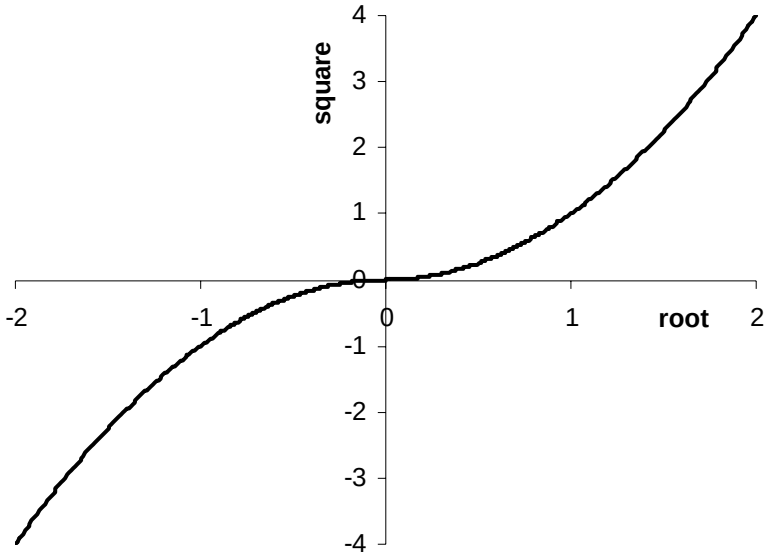

Figure 8-3: Graphical representation of the equation modeled by the **square_robust** class.

This approach makes the resulting function valid over the entire domain of the real numbers, and not just the positive real numbers. It also prevents the discontinuity at zero for the root inverse that would result from piecing together two square root functions, one for the positive domain and its reverse function for the negative domain. A typical application of the **square_robust** class would be to implement the power law equation that calculates the airflow as a function of the pressure difference across an opening.

The mathematical relation modeled by this class is:

$$\begin{cases} root = \sqrt{|square|}, \; if \;\; square \geq 0 \\ root = -\sqrt{|square|}, \; if \;\; square < 0 \end{cases} \tag{8.1}$$

The inverse for the port **square** is expressed in explicit form by:

$$\begin{cases} square = root \cdot root, \; if \;\; root \geq 0 \\ square = -root \cdot root, \; if \;\; root < 0 \end{cases} \tag{8.2}$$

This is a clear case where using a residual inverse to calculate the square root is more efficient because it allows to express the inverse equation in a numerically more robust functional form. The inverse for the port **root** is expressed in residual form by:

$$0 = sign(root) \cdot (root \cdot root) - square \tag{8.3}$$

The residual form enforces the exact same mathematical relation between the variables *square* and *root* as expressed in Equation (8.1) when it is transformed with the root square operator. It is a more robust expression because the partial derivative with respect to *square* no longer tends to infinity when the value of *square* approaches zero. The resulting numerical behavior is better suited for solution with the Newton-Raphson method, which is typically the way strongly-connected components are solved in *SPARK*. Of course, this matters only if the inverse for the port **root** is defined as part of a strongly-connected component and if the variable connected to the port **square** is selected as a break variable.

To compare the partial derivatives for each functional form, we first have to express the Equation (8.1) in residual form, as solved in the Newton-Raphson iteration:

$$F_1(root, square) = \begin{cases} \sqrt{|square|} - root, \; if \;\; square \geq 0 \\ -\sqrt{|square|} - root, \; if \;\; square < 0 \end{cases} \tag{8.4}$$

Then the partial derivative for Equation (8.4) with respect to the variable *square* is:

$$\frac{\partial F_1}{\partial square} = \begin{cases} \dfrac{1}{2\sqrt{|square|}} - \dfrac{\partial root}{\partial square}, \; if \;\; square \geq 0 \\ \dfrac{-1}{2\sqrt{|square|}} - \dfrac{\partial root}{\partial square}, \; if \;\; square < 0 \end{cases} \tag{8.5}$$

When the variable *square* tends to zero, the partial derivative for F_1 will tend to infinity as observed in Equation (8.5). This compares to the partial derivative obtained for the functional form F_2 of Equation (8.3):

$$\begin{aligned} F_2(root, square) &= sign(root) \cdot (root \cdot root) - square \\ \frac{\partial F_2}{\partial square} &= sign(root) \cdot \frac{\partial root}{\partial square} \cdot 2 \cdot root - 1 \end{aligned} \tag{8.6}$$

Clearly, the partial derivative for F_2 is numerically better as there is no longer a problematic division by zero when the variable *square* is equal to zero.

8.7.4 Class Definition

A residual inverse is defined in the `FUNCTIONS {…}` block by using the `RESIDUAL` keyword after the equal sign introducing the `EVALUATE` callback function.

```
FUNCTIONS {
   root    = RESIDUAL square_robust__root( root, square )  ;
   square  = square_robust__square( root ) ;
}
```

The **square_robust** class defines two inverses, one for each port. For each inverse only the `EVALUATE` callback function is specified.

- The inverse assigned to the port **root** is defined as a residual inverse by adding the RESIDUAL keyword in front of the name of the EVALUATE callback `square_robust__root`.

- The inverse assigned to the port **square** is by default an inverse in explicit form since no keyword is specified in front of the name of the EVALUATE callback `square_robust__square`.

Note that the target port **root** appears also as an argument port in the EVALUATE callback `square_robust__root` as required by any inverse expressed in residual form. If you omit to declare the target port(s) also as argument port(s) of the EVALUATE callback for a residual inverse, it will generate a parsing error.

8.7.5 Inverse Function Definition

The EVALUATE callback `square_robust__root` implements the residual inverse for the target port **root** by calculating the residual value using the Equation (8.3). Finally, the residual value is returned using the preprocessor macro RETURN. Here the returned value must be the residual value for the EVALUATE callback. This is to contrast with an EVALUATE callback expressed in explicit form, whereby the result value of the explicit functional form is returned (See Section 3.2.5).

```
EVALUATE( square_robust__root )
{
   ARGDEF(0, root);
   ARGDEF(1, square);

   double residual = SPARK::sign(root)*pow(root, 2.0) - square;

   RETURN( residual );
}
```

8.8 USAGE OF THE DEFAULT RESIDUAL INVERSE IN THE FUNCTIONS STATEMENT

A default residual inverse can be specified at the class level using the DEFAULT_RESIDUAL keyword before the = sign in the FUNCTIONS {…} block, in place of the list of target ports. A default residual inverse must be a single-valued inverse that returns the value of the residual equation modeled by the class. This method is equivalent to populating the atomic class with the same residual inverse for these ports with no inverse yet.

By defining a default inverse for the ports for which no inverse is explicitly specified, the default residual inverse mechanism provides the matching algorithm in *setupcpp* with alternatives in case no complete matching can be obtained with the normal inverses (See Section 12.2).

The following rules apply to default residual inverses.

- A default residual inverse is not assigned to any target ports in the FUNCTIONS {…} statement of the atomic class.

- The argument list of the EVALUATE callback must mention every single port defined in the class to ensure correct variable dependency during the graph analysis.

- A default residual inverse cannot define a PREDICT callback as it very unlikely that the same predictor function can apply to each port defined in the class.

- A default residual inverse must be a single-valued inverse in order to be a valid default inverse that can be assigned to each individual port defined at the interface. Therefore it can be used only in single-valued classes.

- The default residual inverse is defined with a default match level of 1 to make it the least likely alternative to choosing any other “dedicated” inverse specified in the class.

- The default residual mechanism is supported only with the DEFAULT and INTEGRATOR atomic classes, since the SINK atomic classes cannot define EVALUATE callbacks.

The following code snippet shows an atomic class that defines an inverse in explicit form for the port **x** and a default residual inverse for the other ports **y** and **z**. The class models a mathematical expression that does not lend itself very well to finding closed-form equations for each variable through symbolic manipulation.

```
#ifdef SPARK_PARSER

PORT x;
PORT y;
PORT z;

EQUATIONS {
    x = exp(y)*cos(1/z) ;
}

FUNCTIONS {
   DEFAULT_RESIDUAL = default_residual( x, y, z );
   x = inverse__x( y, z );
}

#endif /* SPARK_PARSER */
#include "spark.h"

EVALUATE(inverse__x)
{
   ARGDEF(0, y);
   ARGDEF(1, z);
   double result = exp(y)*cos(1.0/z);
   RETURN( result );
}
EVALUATE(default_residual)
{
   ARGDEF(0, x);
   ARGDEF(1, y);
   ARGDEF(2, z);
   double residual = x - exp(y)*cos(1.0/z);
   RETURN( residual );
}
```

Clearly, it is difficult to produce closed-form inverses for the port **y** and **z** that are expressed in explicit form and that are numerically robust. Instead, we define a default residual inverse with the EVALUATE callback default_residual. Note that all the ports of the class appear in the argument list of the callback. This default inverse will be assigned to either the port **y** or the port **z**, were they to be selected during the matching phase of the *setupcpp* program.

The EVALUATE callback of a DEFAULT_RESIDUAL inverse is implemented in the same manner as the EVALUATE callback expressed in residual form for a single-valued inverse, whereby the residual value is returned using the RETURN preprocessor macro (See Section 8.7).

9 THE CALLBACK FRAMEWORK

9.1 OVERVIEW AND TERMINOLOGY

An important modeling feature of *SPARK* is to be able to associate private data with each inverse comprising an atomic class in order to hold state information or temporary calculations between successive calls. The internal mechanism that enables to support private data while providing backward compatibility with *SPARK* 1 is the callback framework.[35]

The callback framework adds object-oriented capabilities to *SPARK* by providing data encapsulation and polymorphic behavior for each inverse through a collection of callback functions implemented as *C++* free functions (i.e., normal *C* functions). Each callback function is invoked by the *SPARK* solver engine at predetermined points of the simulation task, therefore allowing the user to implement specific operations for each inverse beside calculating the values of the target port(s).

9.1.1 Inverse Type

An atomic class consists of a set of inverses. Each inverse implements a directed data flow through the port interface. The ports assigned to an inverse are called the target ports. Clearly, each inverse in a class is associated with a set of mutually exclusive target ports. In turn, an inverse consists of a set of callback functions. The computational graph produced by the *setupcpp* program is derived from the topological information described by the EVALUATE callback functions.

Single-valued inverse

A single-valued inverse returns the value for one target port only. At most one single-valued inverse can be specified for each port defined at the interface of the atomic class. All inverses defined in *SPARK* 1 were single-valued inverses.

Multi-valued inverse

A multi-valued inverse returns the values for more than one port simultaneously. Only one inverse can be specified for an atomic class that defines a multi-valued inverse. This limitation facilitates the matching operation performed by *setupcpp* by forcing the data flow through the directed multi-valued objects. Future versions of *SPARK* might support more than one multi-valued inverse per class as long as they are each assigned to mutually exclusive sets of ports.

Default residual inverse

A default residual inverse can be defined at the class level. A default residual inverse must be a single-valued inverse that returns the value of the residual equation for the class. It provides an default inverse for the ports for which no inverse is explicitly specified, thus providing the matching algorithm in *setupcpp* with alternatives in case of incomplete matching with the other inverses.

9.1.2 Inverse Instance

A problem consists of a collection of inverse instances defined during the matching operation in *setupcpp*. One inverse instance is defined for each occurrence of a class in the problem definition. There are as many inverse instances as there are atomic objects in the problem.

[35] The callback framework is an extension of the EVALUATE and PREDICT functions defined for each inverse in *SPARK* 1.

Evaluating the collection of inverses for the variables connected to the matched ports in each class solves the problem for the unknown variables. The inverse instances are invoked in a specific order that implements the solution sequence derived by *setupcpp*. The ordered set of inverse instances is also decomposed in independent components identified by a topological sort of the directed computational graph.

An inverse is represented internally in the *SPARK* solver with the `TInverse` class whereas an inverse instance is represented with the `TObject` class. We also refer to the instance of an inverse as an object.

9.1.3 Callback Function

An inverse consists of a collection of callback functions. A callback function is implemented as a simple *C++* free function with a predetermined prototype depending on the type of the callback (See Section 9.1.2). Each callback defined in an inverse is identified by a keyword which precedes the callback function name in the `FUNCTIONS {…}` statement of the atomic class (See Section 9.2). Each inverse can define at most one callback of each type. Other restrictions apply for certain callback functions depending on the type of the inverse.

Callback classification

Callback functions belong to one of the following categories.

Table 9-1: Callback categories and function types.

Callback Category	Callback Function Types	Instance Callback	Static Callback
structors	construct	yes	yes
	destruct	yes	yes
modifier	evaluate	yes	no
	predict	yes	no
non-modifier	prepare step	yes	yes
	commit	yes	yes
	rollback	yes	yes
predicate	check integration step	yes	yes

Detailed description of the callback functions in each category is provided in the Sections 9.4, 9.5, 9.6 and 9.7.

Static and instance callbacks

We distinguish between static callbacks and instance callbacks. Static callbacks apply to an inverse type, whereas instance callbacks apply to each particular instance of an inverse. Static callbacks defined for an inverse are invoked only once whereas the instance callbacks are invoked for each instance of the inverse in question.

Using the analogy with the *C++* programming language, static callbacks can be viewed as the static methods of the inverse class, whereas the instance callbacks can be viewed as the methods of the object instantiated from this inverse class. For example, if there are `N` instances of an inverse, then the static callbacks will be invoked only once per simulation step but the instance callbacks will be invoked for each instance, namely `N` times.

Typically, static callbacks deal with managing private data that is shared by all instances of the same inverse. Note that static modifier callbacks cannot be defined for an inverse. Also, static callbacks cannot have arguments.

9.1.4 Private Data

It is possible to associate private data with each inverse instance by implementing appropriate data management through the different callbacks that the user can define for each inverse. In particular, the structor callbacks can be used to allocate and deallocate the private memory required by each inverse instance. *SPARK* provides the user with a data management API[36] defined in the classapi.h header file. It is the user's responsibility to implement the appropriate operations in the corresponding callback functions using this API (See Section 9.8).

9.2 CALLBACK ENTRY POINTS IN SIMULATION LOOP

Figure 9-1 shows the entry points in the simulation loop where the callback functions are invoked to interact with the solution process. The dotted boxes indicate the different phases of the solution process. First, the problem under study is initialized, then it is solved for each step until the simulation end time is reached or some other condition stops the run. Finally, the problem solver is terminated. The gray boxes represent the entry points where the different types of callback functions are invoked.

[36] API stands for Application Programming Interface.

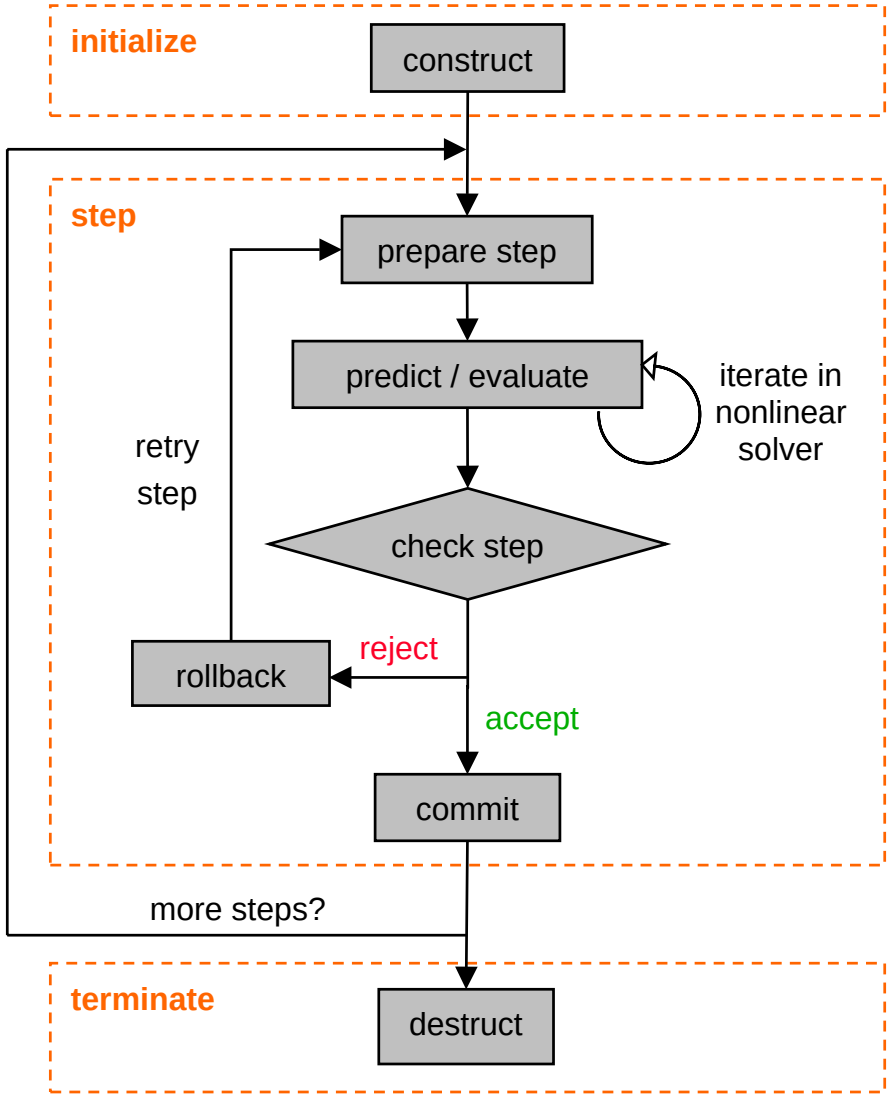

Figure 9-1: Callback entry points in simulation loop.

The evaluate callbacks are invoked when evaluating the components comprised in the problem. The strongly-connected components typically require iterative solution performed in the *SPARK* nonlinear solver. Therefore, the evaluate callbacks might be invoked multiple times over the same step until convergence is obtained in the nonlinear solver. The predict callbacks are invoked only if their target ports are connected to break variables, and they are invoked only once before starting the iterative solution.

The rollback callbacks are invoked only following a rejected step. The commit callbacks are invoked only after an accepted step. The check step entry point is where the check integration step callback functions are invoked whenever the solver is computing a dynamic step.

9.3 Specifying the Callback Functions

9.3.1 The `FUNCTIONS` Statement

Callbacks are declared in the `FUNCTIONS {…}` statement of an atomic class for each inverse using keywords that identify the various callback types. The following code snippet shows the general syntax of the

FUNCTIONS {…} statement where callbacks can be declared either for a default residual inverse or an inverse assigned to target port(s). Elements specified within < > are optional.

```
FUNCTIONS {
   DEFAULT_RESIDUAL = residual_function( port1,...,portN )
           <method1 = method1_function( port2,...)
            method2 = method2_function( ... )
                    ....>
   ;

   port1 <,port2,port3,...> = <RESIDUAL> evaluate_function ( port2,...)
                   <method1 = method1_function( port2,...)
                    method2 = method2_function( ... )
                    ....>
   ;
}
```

Note that the keyword DEFAULT_RESIDUAL indicates the inverse type, and not the callback type, although this inverse type implies that the evaluate callback function be expressed in residual form (See Section 8.8).

9.3.2 Callback Keywords

Here method1, method2... are keywords that uniquely specify the type of the callback function along with the expected prototype of the *C++* function that implements the callback.

Table 9-2: Keywords for the instance callbacks.

Keywords for instance callbacks	Description
EVALUATE	Used to specify a evaluate callback in explicit form (single-valued or multi-valued inverse). Implied if not specified.
RESIDUAL	Used to specify an evaluate callback in residual form (single-valued or multi-valued inverse). Optional.
PREDICT	Used to return predicted values before the first iteration of the nonlinear solver.
CONSTRUCT	Used to specify the construct callback function.
DESTRUCT	Used to specify the destruct callback function.
PREPARE_STEP	Used to specify the prepare step callback function.
COMMIT	Used to specify the commit callback function.
ROLLBACK	Used to specify the rollback callback function.
CHECK_INTEGRATION_STEP	Used to specify the check integration step callback function.

Table 9-3: Keywords for the static callbacks.

Keywords for static callbacks	Description
STATIC_CONSTRUCT	Used to specify the static construct callback function.
STATIC_DESTRUCT	Used to specify the static destruct callback function.
STATIC_PREPARE_STEP	Used to specify the static prepare step callback function.
STATIC_CHECK_INTEGRATION_STEP	Used to specify the static check integration step callback function.
STATIC_COMMIT	Used to specify the static commit callback function.
STATIC_ROLLBACK	Used to specify the static rollback callback function.

Preprocessor macros named after the callbacks keywords are defined in the file spark.h to facilitate the implementation of each callback function by hiding the prototype and argument declarations.

9.4 STRUCTOR CALLBACKS

As described in detail in Section 9.8, the structor callbacks are used to allocate and deallocate memory for private data:

- specific to each inverse instance using the CONSTRUCT and DESTRUCT callbacks, or
- shared by all inverse instances using the STATIC_CONSTRUCT and STATIC_DESTRUCT callbacks.

Constructor callbacks are called before solving the first step during the problem initialization phase and destructor callbacks are called after solving the last step during the problem termination phase (See Figure 9-1). One time initialization of private data can also be performed in the constructor callbacks.

9.4.1 Syntax

Structor callbacks cannot modify the solution values of the target ports associated with the inverse.

The structor callback functions are specified with the following keywords in the FUNCTIONS {…} statement of the atomic class. Homonymous preprocessor macros can be used to implement the *C++* functions for each structor callback.

Table 9-4: Structor callbacks.

Callback Type	Callback Keyword	C++ Function Prototype
construct	CONSTRUCT	`void f(TObject* object, ArgList args)`
	STATIC _CONSTRUCT	`void f(TInverse* inverse)`
destruct	DESTRUCT	`void f(TObject* object, ArgList args)`
	STATIC _DESTRUCT	`void f(TInverse* inverse)`

9.4.2 Rules

- If the CONSTRUCT callback is defined then the DESTRUCT callback must also be provided and vice-versa to ensure that memory allocated in the constructor callbacks is properly freed in the destructor callback.

- If the STATIC_CONSTRUCT callback is defined then the STATIC_DESTRUCT callback must also be provided and vice-versa to ensure that memory allocated in the static constructor callbacks is properly freed in the static destructor callback.

Table 9-5: Keyword table by inverse type for the structor callbacks.

Inverse Types			Structor Callbacks			
			instance construct	static construct	instance destruct	static destruct
sink			CONSTRUCT	STATIC_CONSTRUCT	DESTRUCT	STATIC_DESTRUCT
integrator			CONSTRUCT	STATIC_CONSTRUCT	DESTRUCT	STATIC_DESTRUCT
default	single-valued	explicit form	CONSTRUCT	STATIC_CONSTRUCT	DESTRUCT	STATIC_DESTRUCT
		residual form	CONSTRUCT	STATIC_CONSTRUCT	DESTRUCT	STATIC_DESTRUCT
		default residual	CONSTRUCT	STATIC_CONSTRUCT	DESTRUCT	STATIC_DESTRUCT
	multi-valued	explicit form	CONSTRUCT	STATIC_CONSTRUCT	DESTRUCT	STATIC_DESTRUCT
		residual form	CONSTRUCT	STATIC_CONSTRUCT	DESTRUCT	STATIC_DESTRUCT

9.5 MODIFIER CALLBACKS

Modifier callbacks are used to compute the value(s) of the target port(s) assigned to the inverse. Modifiers are the only callback functions that can return solution values for the target ports. They are called by the *SPARK* solver at each time step, possibly many times in order to obtain convergence.

There are two types of modifier callbacks: the EVALUATE callback and the PREDICT callback. The EVALUATE callback "evaluates" the object to produce the values for the target ports. The PREDICT callback is only invoked if the associated port(s) is connected to a break variable in a strongly-connected component. Then the PREDICT callback produces the predicted values for the target ports before starting the iterative solution process.

There can be no static modifier callbacks because static callbacks are not assigned to any target ports in particular but refer to all instances of the same inverse.

9.5.1 Syntax

The evaluate callback can be implemented in two different forms:

- The **explicit** form specified with the optional keyword EVALUATE returns the solution value(s) of the inverse equation(s) for the associated target port(s) as a `double`.
- The **residual** form specified with the keyword RESIDUAL returns the value(s) of the residual equation(s) for the associated target port(s) as a `double`.

The values returned by the modifier callback functions are copied to the list of target ports implemented with type `ResList`.

Table 9-6: Modifier callbacks.

Callback Type	Callback Keyword	C++ Function Prototype
evaluate	EVALUATE	`void f(TObject* object, ArgList args, ResList` **`results`**`)`
	RESIDUAL	`void f(TObject* object, ArgList args, ResList` **`residuals`**`)`
predict	PREDICT	`void f(TObject* object, ArgList args, ResList` **`predictors`**`)`

9.5.2 Rules

- The inverse for a sink atomic class defines neither an evaluate callback nor a predict callback since it cannot return any solution values.
- All other atomic classes must define an evaluate callback. The predict callback is optional.
- To enforce the proper data dependency in the graph-theoretic processing, a residual evaluate callback (i.e., defined with the keyword RESIDUAL) must also declare in its argument list the target port(s) it is associated with.

  In the next example we show a `FUNCTION {…}` statement for a residual inverse where the target ports `o1` and `o2` are correctly specified in the argument list.

```
PORT o1;
PORT o2;
PORT i1;
PORT i2;
FUNCTIONS {
   o1,o2 = RESIDUAL residual_fn(i1,i2,o1,o2 ) ;
```

- The default residual callback (i.e., the evaluate callback defined with a default inverse) must be a single-valued inverse. Also, it cannot define a predict callback and all the class ports must appear in the argument list.

  In the next example we show a `FUNCTION {…}` statement for a default residual inverse where all class ports are correctly specified in the argument list.

```
PORT o1;
PORT o2;
PORT i1;
PORT i2;
FUNCTIONS {
   DEFAULT_RESIDUAL = default_residual_fn(i1,i2,o1,o2 ) ;
   …
}
```

Table 9-7 shows the list of keywords by inverse type for the modifier callbacks. Keywords within parenthesis indicate that the keywords are implied if not explicitly specified in the `FUNCTION {…}` statement. The only such keyword is the `EVALUATE` keyword that declares an evaluate callback function in explicit form. This is the default callback that is always specified for the `DEFAULT` classes.

Table 9-7: Keyword table by inverse type for the modifier callbacks.

Inverse Types			Modifier Callbacks	
			instance predict	instance evaluate
sink			no	no
integrator			PREDICT	(EVALUATE)
default	single-valued	explicit form	PREDICT	(EVALUATE)
		residual form	PREDICT	RESIDUAL
		default residual	no	DEFAULT_RESIDUAL
	multi-valued	explicit form	PREDICT	(EVALUATE)
		residual form	PREDICT	RESIDUAL

9.6 Non-Modifier Callbacks

The non-modifier callbacks deal with private data management at different phases of each time step. They are invoked in the stepping method of the problem simulator.

Non-modifier callback functions can be specified for each inverse type (static callbacks) or for each inverse instance (instance callbacks).

- The prepare step callback is invoked before starting the evaluation of each step, therefore allowing the user to prepare the private data for the current step.
- The commit callback is invoked when the current step has been accepted, therefore allowing the user to update the private data of the inverse for the next step.
- The rollback callback is invoked when the current step has been rejected, therefore allowing the user to reset the private data like it was at the beginning of the current step, thus allowing for a fresh step.

9.6.1 Syntax

Non-modifier callbacks cannot modify the solution values of the target ports associated with them. Hence their name.

Table 9-8: Non-modifier callbacks.

Callback Type	Callback Keyword	C++ Function Prototype
prepare step	PREPARE_STEP	void f(TObject* object, ArgList args)
	STATIC_PREPARE_STEP	void f(TInverse* inverse)
commit	COMMIT	void f(TObject* object, ArgList args)
	STATIC_COMMIT	void f(TInverse* inverse)
rollback	ROLLBACK	void f(TObject* object, ArgList args)
	STATIC_ROLLBACK	void f(TInverse* inverse)

9.6.2 Rules

None of the non-modifier callbacks are required. No specific rules need to be enforced.

Table 9-9: Keyword table by inverse type for the non-modifier callbacks.

Inverse types			Non-modifier Callbacks					
			instance prepare step	static prepare step	instance commit	static commit	instance rollback	static rollback
sink			PREPARE_STEP	STATIC_PREPARE_STEP	COMMIT	STATIC_COMMIT	ROLLBACK	STATIC_ROLLBACK
integrator			PREPARE_STEP	STATIC_PREPARE_STEP	COMMIT	STATIC_COMMIT	ROLLBACK	STATIC_ROLLBACK
default	single-valued	explicit form	PREPARE_STEP	STATIC_PREPARE_STEP	COMMIT	STATIC_COMMIT	ROLLBACK	STATIC_ROLLBACK
		residual form	PREPARE_STEP	STATIC_PREPARE_STEP	COMMIT	STATIC_COMMIT	ROLLBACK	STATIC_ROLLBACK
		default residual	PREPARE_STEP	STATIC_PREPARE_STEP	COMMIT	STATIC_COMMIT	ROLLBACK	STATIC_ROLLBACK
	multi-valued	explicit form	PREPARE_STEP	STATIC_PREPARE_STEP	COMMIT	STATIC_COMMIT	ROLLBACK	STATIC_ROLLBACK
		residual form	PREPARE_STEP	STATIC_PREPARE_STEP	COMMIT	STATIC_COMMIT	ROLLBACK	STATIC_ROLLBACK

9.7 PREDICATE CALLBACKS

The predicate callbacks return a boolean value to the solver that is interpreted to decide whether to accept the current step or reject it.

In the current version, only INTEGRATOR classes can specify predicate callbacks, namely the callback used to check the integration step. Returning false means that the current step must be rejected, whereas returning the value true means that the current step should be accepted.

The check integration step callback is invoked after the current step has been successfully computed, therefore allowing the user to accept or reject the current solution using solver requests. If the step is accepted then the commit callbacks will be invoked next before going to the next step. If the step is rejected then the rollback callbacks will be invoked before re-trying the same step (probably with a different time step though!).

9.7.1 Syntax

Predicate callbacks cannot modify the solution values of the target ports.

Table 9-10: Predicate callbacks.

Callback Type	Callback Keyword	C++ Function Prototype
check integration step	`CHECK_INTEGRATION_STEP`	`bool f(TObject* object, ArgList args)`
	`STATIC_CHECK_INTEGRATION_STEP`[37]	`bool f(TInverse* inverse)`

9.7.2 Rules

Only integrator classes (i.e., classes define with `CLASSTYPE INTEGRATOR`) can define the check integration step callbacks.

The only predicate callback function is the check integration step callback. It can be defined only for `INTEGRATOR` classes.

Table 9-11: Keyword table by inverse type for the predicate callbacks.

Inverse types			Predicate Callbacks: instance check integration step	Predicate Callbacks: static check integration step
sink			no	no
integrator			`CHECK_INTEGRATION_STEP`	`STATIC_CHECK_INTEGRATION_STEP`
default	single-valued	explicit form	no	no
		residual form	no	no
		default residual	no	no
	multi-valued	explicit form	no	no
		residual form	no	no

[37] Unlike the way if shows in the table the callback keyword `STATIC_CHECK_INTEGRATION_STEP` is one word.

9.8 DEFINING PRIVATE DATA FOR AN INVERSE

This section explains how to define private data for an inverse using the callback mechanism. Private data attached to an inverse instance, also referred to as an object, is called **instance private data** as it is unique to each instance of an inverse. Private data attached to an inverse type is called **static private data** as it is shared by all instances of the same inverse type.

Instance static data is allocated in the `CONSTRUCT` callback and is deallocated in the `DESTRUCT` callback. Static private data is allocated in the `STATIC_CONSTRUCT` callback and is deallocated in the `STATIC_DESTRUCT` callback.

9.8.1 Private Data Mechanism

The atomic classes in *SPARK* are not implemented as *C++* classes, therefore it is not possible to rely on the *C++* language to support the private data mechanism as a built-in feature. The rationale behind this design decision was to avoid the performance penalty incurred by the virtual function call mechanism that would be needed to provide the desired polymorphic behavior for each atomic class.

Instead, the atomic class "methods" are implemented as *C++* free functions, the callback functions, specified for each inverse. The private data mechanism is then enabled by passing pointers to the *SPARK* classes, which describe each inverse type and its instances, to the callback functions. Internally to the *SPARK* solver, an inverse type is described with the `TInverse` class, whereas an inverse instance is defined with the `TObject` class. Each inverse type is uniquely identified at runtime through the instance of the `TInverse` class attached to the static callbacks. Similarly, each inverse instance is uniquely identified at runtime through the instance of the `TObject` class attached to the instance callbacks.

The htm/chm tutorial *SPARK Atomic Class API* should be consulted for more information on the `TInverse` and `TObject` classes.

Instance Private Data

In order to encapsulate private data with each inverse instance, *SPARK* defines a `void` pointer in the `TObject` class that describes an inverse instance in the solver. This `void` pointer can be used to store the address of any data[38] that is used to implement the instance callbacks defined for this inverse. The pointer to the `TObject` class describing each inverse instance is then passed as the first argument to each instance callback function. See the instance callback prototypes in Table 9-4, Table 9-6, Table 9-8 and Table 9-10.

Static Private Data

Similarly, *SPARK* defines a `void` pointer in the `TInverse` class that describes an inverse type in the solver. This `void` pointer can be used to store the address of any data that is used to implement the static callbacks defined for this inverse. The pointer to the `TInverse` class describing each inverse is then passed as the first argument to each static callback function. See the static callback prototypes in Table 9-4, Table 9-6, Table 9-8 and Table 9-10.

[38] By data we refer here to an instance of any *C++* type, such as a class, a struct, a function, or any fundamental type.

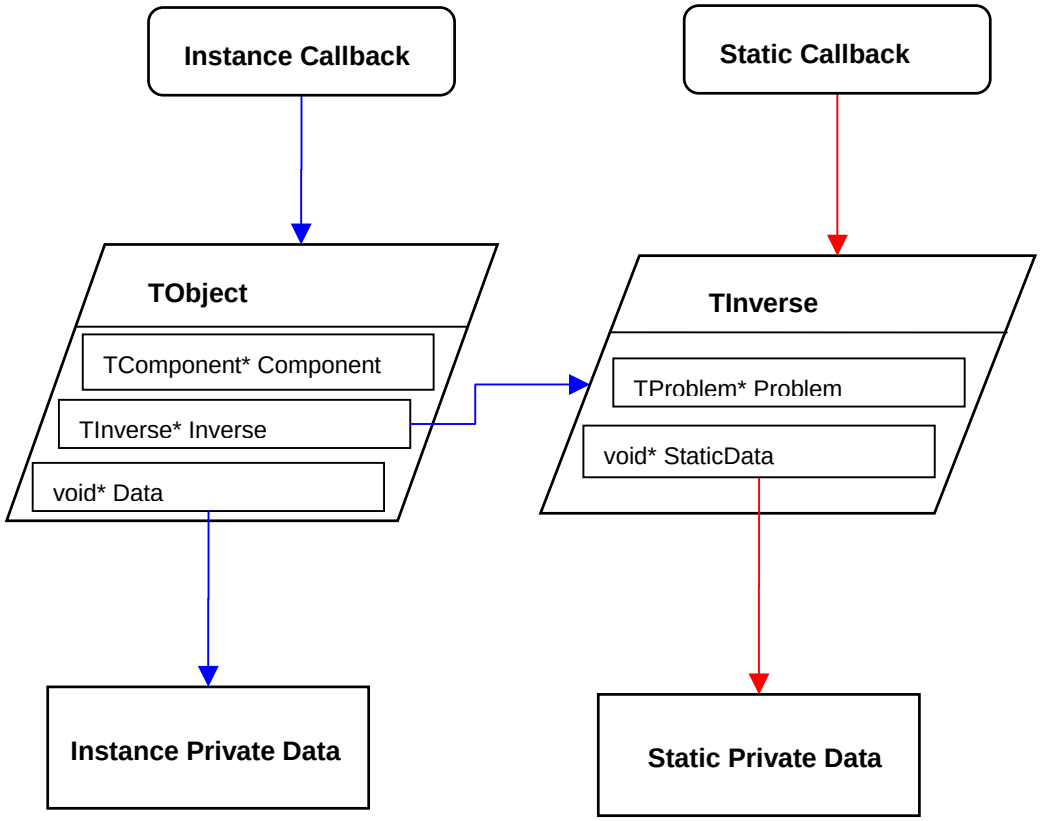

Figure 9-2: Schematic representation of the relationships between callbacks and private data.

Figure 9-2 shows the `void` pointers defined in the `TObject` and `TInverse` classes that point to the areas in memory where the instance private data and static private data are stored. The blue arrows indicate the data flow from the instance callbacks, whereas the red arrows indicate the data flow from the static callbacks.

The pointer to the `TInverse` class is also available from the `TObject` classes that represent the instances of this inverse. This makes it possible to access the `void` pointer to the static private data from each instance callback.

The “this” Pointer

The `void` pointers stored in the `TObject` and `TInverse` classes essentially model the `this` pointer paradigm inherent to the concept of classes in the *C++* language. The `TObject` and `TInverse` classes merely serve to distinguish between the instance of an inverse and the type of an inverse. Thus, instance private data corresponds to the member data of a *C++* class, whereas static private data corresponds to the member data defined as `static` in a *C++* class.

The `TObject` and `TInverse` classes define get/set methods to operate on the `void` pointers they each define as member data. The set method is used to store the address of the newly allocated private data. The get method is used to retrieve the address of the private data that was stored previously with the set method. Table 9-12 and Table 9-13 show the prototypes of the get/set methods defined in the `TObject` and `TInverse` classes.

Table 9-12: Get/set methods of the `TObject` class.

Method	Description
`void  TObject::SetData(void* data)`	Stores the address to the instance private data in the `void` pointer member data.
`void* TObject::GetData()`	Retrieves the address of the instance private data as a `void` pointer.

Table 9-13: Get/set methods of the `TInverse` class.

Method	Description
`void  TInverse::SetStaticData(void* data)`	Stores the address to the static private data in the `void` pointer member data.
`void* TInverse::GetStaticData()`	Retrieves the address of the static private data as a `void` pointer.

Note that the names of the get/set methods differ between the `TObject` and `TInverse` classes in order to distinguish at compile time between operations intended on instance private data or on static private data. This naming convention avoids possible confusions due to the lack of available type information implied by the use of the `void` pointer. Indeed, the *C++* compiler will complain if you try to retrieve instance private data with the `GetData()` method from a static callback that carries the pointer to a `TInverse` class.

The address of the private data is implemented as a `void` pointer in the *SPARK* classes in order to achieve maximum generality in terms of what type of *C++* objects can be used to represent private data. This results in the get method returning the address of an un-typed *C++* object. Therefore, the retrieved address must be cast to a pointer to the original type of the *C++* object used to implement this specific private data. It is the user's responsibility to perform this type cast in the body of the callback functions whenever the `void` pointer is retrieved.

If `Object` represents a pointer to a `TObject` class, the `void` pointer is cast to the original type `TData` using the *C++* construct `static_cast< >` as shown in the following code snippet.

```
   TData* MyData = static_cast<TData*>( Object->GetData() );
```

If `Inverse` represents a pointer to a `TInverse` class, the `void` pointer is cast to the original type `TStaticData` using the *C++* construct `static_cast< >` as shown in the following code snippet.

```
   TStaticData* MyData = static_cast<TStaticData*>( Inverse->GetStaticData() );
```

Note that no type casting is required when using the set methods.

Preprocessor Macros

Preprocessor macros are used to hide the implementation details pertaining to the callback function prototypes in the *SPARK* atomic classes. Compatible preprocessor macros are defined in the spark.h file to operate on static private data and instance private data from within the body of the callback functions.

Table 9-14: List of preprocessor macros operating on static private data.

Macro	Description
`THIS`	Returns the pointer to the `TInverse` class this static callback is defined for.
`ACTIVE_PROBLEM`	Returns the pointer to the `TProblem` class the current inverse belongs to.

Table 9-15: List of preprocessor macros operating on instance private data.

Macro	Description
THIS	Returns the pointer to the TObject class this instance callback is defined for.
ACTIVE_PROBLEM	Returns the pointer to the TProblem class the current inverse instance belongs to.
ACTIVE_INVERSE	Returns the pointer to the TInverse class this instance callback belongs to.
ACTIVE_COMPONENT	Returns the pointer to the TComponent class the current inverse instance belongs to.

Using the ACTIVE_INVERSE macro in an instance callback returns the pointer to the TInverse class describing this inverse type. This pointer can then be used to operate on the void pointer that stores the static private data, thus granting access to static private data from the instance callbacks.

The htm/chm tutorial *SPARK Atomic Class API* should be consulted for more information on the TProblem and TComponent classes.

9.8.2 Example of an Inverse with Private Data

The atomic class **analytical_frst_ord** uses an instance of the class TData to compute the analytical solution of a first order homogenous ODE with constant, linear coefficients. Since the ODE coefficients are assumed to remain constant for the simulation run, it is possible to derive the equations for the analytical solution of the dynamic variable and its time derivative at the onset of the simulation, when the coefficient values are first known at runtime.

The **analytical_frst_ord** class defines a multi-valued inverse that returns the values of both the dynamic variable, assigned to the port **x**, and its time-derivative, assigned to the port **xdot**, as a function of the current time, assigned to the port **time**. Along with the multi-valued EVALUATE callback, the inverse defines a CONSTRUCT callback and a DESTRUCT callback where the private data management is performed.

The values for the analytical solution are actually computed by the instance of the class TData that is attached as instance private data to the multi-valued inverse in the CONSTRUCT callback. The methods TData::x() and TData::xdot() are called in the EVALUATE callback to return the values of the analytical solution at the current time for the dynamic variable and its time-derivative. Finally, the instance private data that was allocated in the CONSTRUCT callback is deallocated in the DESTRUCT callback.

The class also defines two ports **A** and **B** for the ODE coefficients, as well as another port **x_IC** for the initial value of the dynamic variable. These ports are used only in the CONSTRUCT callback to initialize the TData instance. Note that the port **time** is also mentioned in the CONSTRUCT callback to obtain the value of the initial time.

```
/// analytical_frst_ord.cc
/// Atomic class that computes the analytical solution (a.k.a. closed-form solution)
/// of the first-order, constant coefficient, linear, homogeneous ODE.
///
/// ODE:
///xdot = B - A*x
///
/// Initial conditions:
///x(t_IC) = x_IC
///
/// Analytical solution: (can be used to compute the true integration error)
```

```
/// For more details consult
/// http://oregonstate.edu/dept/math/CalculusQuestStudyGuides/ode/first/linear/linear.html
///
/// x(t) = C * exp(-A*(t-t_IC)) + B/A
/// where:
///   C + b/a = x_IC
///  b/a = lim x (t->inf)
///
/// where :
///    x    : dynamic variable
///    xdot : first derivative of x
///
///    A   : constant coefficient
///    B   : constant coefficient
///
////////////////////////////////////////////////////////////////////////////////

#ifdef SPARK_PARSER

PORT time "time" [s];
PORT A    "A";
PORT B    "B";
PORT x    "x";
PORT xdot "xdot";
PORT x_IC "initial condition for x";

EQUATIONS {
   x, xdot = analytical_frst_ord( time, x_IC, A, B );
}

FUNCTIONS {
   x, xdot = analytical_frst_ord__evaluate( time )
      CONSTRUCT = analytical_frst_ord__construct( time, x_IC, A, B )
      DESTRUCT  = analytical_frst_ord__destruct( )
      ;
}

#endif  /*SPARK_PARSER*/


#include <strstream>
using std::ostrstream;
using std::ends;

#include "spark.h"


// Class that calculates the analytical solution of a 1st-order ODE with constant,
// linear coefficients. The equations describing the analytical solution are constructed
// in the class constructor.
class TData {
public:
   // Structors
   TData(double t_IC, double x_IC, double a, double b)
      : T_IC(t_IC), C(x_IC - B/A), A(a), B(b)
   {}
   ~TData()
   {}

   // Main methods
   double x(double time) { return C * exp(-A*(time-T_IC)) + B/A; }
   double xdot(double time) { return -A * C * exp(-A*(time-T_IC)); }

   double GetA() const { return A; }
   double GetB() const { return B; }
   double GetC() const { return C; }
   double GetT_IC() const { return T_IC; }

private:
   // Private data
```

```
   double T_IC;
   double A ;
   double B ;
   double C;
};


EVALUATE( analytical_frst_ord__evaluate )
{
   ARGUMENT( 0, time );
   TARGET( 0, x );
   TARGET( 1, xdot );

   // Cast void* to private data type
   TData* MyData = static_cast<TData*>( THIS->GetData() );

   // Set target values
   x    = MyData->x( time );
   xdot = MyData->xdot( time );
}

CONSTRUCT( analytical_frst_ord__construct )
{
   ARGUMENT( 0, time );
   ARGUMENT( 1, x_IC );
   ARGUMENT( 2, A );
   ARGUMENT( 3, B );

   // Allocate instance private data
   TData* MyData = new TData( time, x_IC, A, B );

   // Check whether memory was allocated correctly or not
   if ( MyData == 0 ) {
      REQUEST__ABORT( "Could not allocate instance private data!" );
   }

   // Store pointer to private data within this object
   THIS->SetData( MyData );

   // Report equation string for analytical solution to error log file
   ostrstream Text;
   Text << "x(t) = " << MyData->GetC() << " * exp(" << MyData->GetA()
        << "*( time - " << MyData->GetT_IC() << ") ) + "
        << MyData->GetB()/MyData->GetA() << ends;

   ERROR_LOG( Text.str() );
}

DESTRUCT( analytical_frst_ord__destruct )
{
   // Cast void* to private data type
   TData* MyData = static_cast<TData*>( THIS->GetData() );

   // Release allocated memory
   if ( MyData ) {
      delete MyData;
   }
}
```

Allocate and Attach Private Data in `CONSTRUCT` Callback

The following code snippet from the `CONSTRUCT` callback shows how the instance private data, described by the *C++* class `TData`, is first allocated on the heap using the *C++* operator `new` and then attached to the pointer to the `TObject` class by invoking the `SetData()` method.

```
   // Allocate instance private data
   TData* MyData = new TData( time, x_IC, A, B );
```

```
// Check whether memory was allocated correctly or not
if ( MyData == 0 ) {
   REQUEST__ABORT( "Could not allocate instance private data!" );
}

// Store pointer to private data within this object
THIS->SetData( MyData );
```

Before storing the instance private data within this object, we verify that the allocation operation succeeded by checking the returned address `MyData` of the allocated instance private data. If it failed, we send a request to abort the simulation (See Section 10.2). This is good practice as it prevents memory access problems when the pointer is dereferenced later.

Deallocate and Detach Private Data in `DESTRUCT` Callback

The following code snippet from the `DESTRUCT` callback shows how the instance private data, described by the *C++* class `TData`, is first retrieved from the `TObject` class by invoking the `GetData()` method and then deallocated using the *C++* operator `delete`.

```
// Cast void* to private data type
TData* MyData = static_cast<TData*>( THIS->GetData() );

// Release allocated memory
if ( MyData ) {
   delete MyData;
}
```

Before calling the `delete` operator, the `void` pointer returned by the method `GetData()`is cast to the a pointer to the original type `TData`. This prevents memory leaks as the `delete` operator frees the memory occupied by the original *C++* type.

Retrieve Private Data in `EVALUATE` Callback

The following code snippet from the `EVALUATE` callback shows how the instance private data is retrieved from the `TObject` class by invoking the `GetData()` method. Like in the `DESTRUCT` callback the returned pointer is cast to its original type before being used. Then, the `TData::x()` and `TData::xdot()`methods are invoked to calculate the analytical solution for both target ports using the private data stored with this inverse instance.

```
// Cast void* to private data type
TData* MyData = static_cast<TData*>( THIS->GetData() );

// Set target values
x    = MyData->x( time );
xdot = MyData->xdot( time );
```

10 THE REQUEST MECHANISM

10.1 CONCEPT

Requests can be sent from the callback functions to influence the behavior of the problem simulator at runtime. The *SPARK* solver pools all requests received from the callbacks over the current step. Then, the requests are processed to identify whether they are valid or not depending on the current context of the simulator. Finally, the valid requests are dispatched to the solver managers that will perform the associated operations. Thus, the execution of the actions implied by a request is deferred to the appropriate moments of the simulation.

Some requests hold data that is used to perform the associated task in the solver. Thus, the request mechanism can be viewed as another way to transferring data from the callbacks back to the solver. The other main data exchange between the callbacks and the solver happens through the target values returned from the modifier callbacks and the boolean values returned by the predicate callbacks.

Requests can be classified in four categories:

- Utility requests
- Requests that trigger the state transitions of the simulator
- Requests that support the time event mechanism
- Requests that support the integration process

The request mechanism is also used internally in the solver to perform the associated tasks, such as restarting or aborting the simulation, generating a snapshot or a report, … . This design approach allows to conveniently implement the problem simulator as a finite-state machine, whereby the state transitions are triggered by the execution of the corresponding requests.

Requests sent from callbacks are called external requests, whereas requests sent from the solver are called internal requests. The operation of the request mechanism can be traced at runtime in the run log file using the *SPARK* diagnostic mechanism (See Section 12.5).

Requests are sent from the body of the callback functions using the preprocessor macros specified in the file spark.h. All request macros are prefixed with the string `REQUEST__`. Also, all request macros expect a `const char*` string as the first argument used to identify the calling context. The context string is then used by the diagnostic mechanism.

Note that the graph-theoretic analysis is unaware of which classes send requests since the requests are specified only in the callback functions and not in the class definition. Indeed, requests do not modify the computational graph but only the behavior of the solver at runtime.

10.2 UTILITY REQUESTS

Utility requests trigger actions that do not directly influence the operation of the solver. They are executed only after a successful step following the call to the commit callbacks (See Figure 9-1).

Table 10-1: List of utility requests.

Request name	Preprocessor macro	Description
report	`REQUEST__REPORT( context )`	Generates a report in the output file at the end of the current step. If there is no output file used in this simulation run, then no action is taken.
snapshot	`REQUEST__SNAPSHOT( context, filename )`	Generates a new snapshot file at the end of the current step from the path `filename` specified as `const char*`. If `filename` is not a valid file name, then no snapshot is generated.

10.3 STATE TRANSITION REQUESTS

The state transition requests interact with the finite-state machine that performs the tasks needed to solve the problem under study from the initial time to the final time. Simulating a problem consists in executing the finite-state machine until the end state is reached, whereby successive static and dynamic steps are taken according to the transitions triggered at runtime either by external requests or by the solver's built-in rules. Table 10-2 describes the state transition requests and the actions triggered by them.

Table 10-2: List of state transition requests.

Request name	Preprocessor macro	Description
abort	`REQUEST__ABORT( context )`	Forces to abort the simulation first by terminating the problem and then by exiting the process. It is executed regardless of whether the current step is accepted or not.
stop	`REQUEST__STOP( context )`	Executed only after a successful step. Stops the simulation and returns from the solver normally, albeit before the specified final time.
restart	`REQUEST__RESTART( context )`	Forces the simulator to restart the simulation by solving a single static step (i.e., the global time is not advanced). It is executed only after a successful step.

Figure 10-1 shows the state-machine implementing the solution process. The arrows indicate the state transitions. The dotted, red arrows refer to the state transitions triggered by the requests, whereas the plain, black arrows represent the built-in state transitions enforced by the solver. The rounded boxes designate the different states of the simulator. The lozenge boxes designate conditional rules which trigger different state transitions depending on the boolean value produced by the evaluation of the rule. In particular, the conditional rule labeled **IC?** refers to whether or not to perform an initial consistent calculation as specified in the run-control file (See Section 18). The other conditional rule deals with detecting whether the simulation final time has been reached. Finally, the state labeled **set time step** implements the task of estimating the time step to use for the next dynamic step.

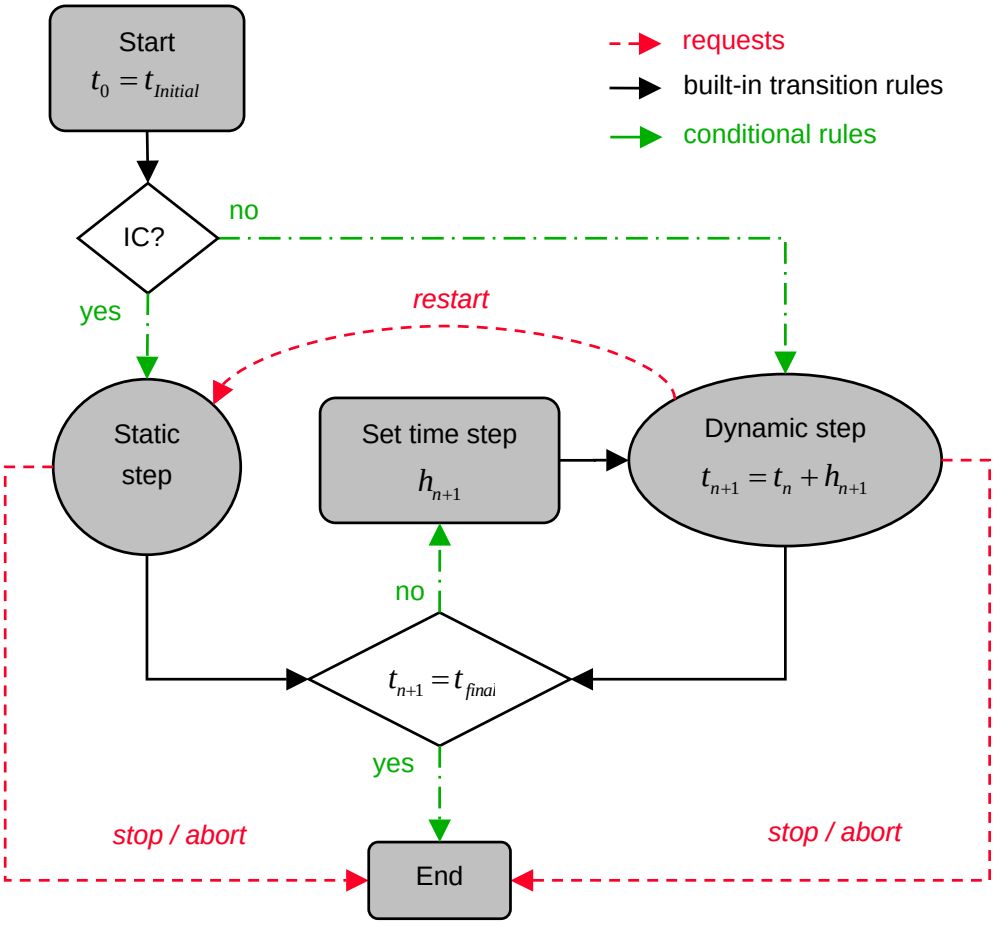

Figure 10-1: State-machine and transitions implementing the simulation process.

10.4 Time Event Requests

The time event requests deal with synchronizing the time-stepping operation in solver with user-specified meeting points. The meeting points are then taken into account when deriving the time step for the next dynamic step.

Table 10-3: List of time event requests.

Request name	Preprocessor macro	Description
set meeting point	`REQUEST__SET_MEETING_POINT( context, time )`	Requests solver to synchronize its global time with the meeting point `time` specified as `double`. If the meeting point does not lie ahead of the current time, then the request will not be processed.
clear meeting points	`REQUEST__CLEAR_MEETING_POINTS( context )`	Clears the list of all meeting points requested so far in solver.

The meeting points can only be synchronized with in the solver if variable time stepping is allowed. If the variable time step mode is not selected in the run-control file (See Section 18), then the time event requests are discarded because the time step cannot be adapted.

Future versions of *SPARK* will define new requests to support the state event mechanism that will add a discrete solving capability to the continuous solver.

10.5 INTEGRATION REQUESTS

The integration events can be requested by `INTEGRATOR` classes only. They provide support for the integration process by interacting with the solver's stepping operation and the time step selection process. The only integration request implemented so far is the request to set the global time step. More requests will be defined in future versions to support integration schemes such as the Runge-Kutta schemes, which require changing the way a dynamic step is performed.

Table 10-4: List of integration requests.

Request name	Preprocessor macro	Description
set time step	`REQUEST__SET_TIME_STEP( context, h )`	If executed following a rejected step, indicates to the solver which time step to use to retry the current dynamic step. If executed after an accepted step, indicates to the solver which time step to use for the next dynamic step. The candidate time step `h` is specified as `double`.

The time step selected by the solver takes into account the smallest candidate time step obtained from processing all the time step requests sent over the current step. Note that the time step requests can only be sent from the commit and rollback callbacks. Due to other constraints, such as synchronizing with meeting points, the actual time step used for the next step could be smaller than the candidate time step, but never larger.

Finally, we note that thanks to the time step request, integrator classes can monitor the integration error and adapt the time step accordingly in order to satisfy the integration error prescribed by the user. The classes **integrator_euler** and **integrator_trapezoidal** located in the globalclass directory are examples of predictor-corrector integration schemes that provide error control through adaptive time step operation.

11 SOLUTION METHOD CONTROLS

While the fundamental, graph-theoretic methodology in *SPARK* is always the same, there are some options you can set to control the actual numerical methods employed. The *VisualSPARK* and *WinSPARK* graphical user interfaces provide menus for setting these options. If you are working at the command line, you can set these options by editing the probName.prf file. However, to explain these options we must first review the fundamental *SPARK* methodology.

11.1 SOLUTION METHODOLOGY

As noted previously, *SPARK* generates a *C++* program to solve the problem expressed in your probName.pr file. To generate this program, graph-theoretic methods are used to decompose the problem into a series of smaller problems, called **components**, that can be solved independently. A component might be a sequence of atomic inverse functions[39] that need to be executed in order; this is the case if no iteration is required in that particular component. On the other hand, iteration may be required, in which case the component, in graph theoretic terms, is a **strongly connected component**. While all equations in a strongly connected component are involved in the iterative solution, usually not all variables need be iterates. Therefore *SPARK* uses graph algorithms to determine a small set of so called **break variables** that break all cycles in the component; these variables constitute a **cut set**.

By default, *SPARK* will attempt to solve each strongly connected component using the Newton-Raphson method, treating the cut set as the vector of independent variables (see Section 2.5). If your problem solves correctly with the default method for the tolerance specified in the global settings, it is probably best not to change it. However, if it fails to solve, it will probably be due to either non-convergence of the Newton-Raphson iteration, or numerical exceptions (i.e., values of problem variables that exceed the capabilities of the computer). In either case, it is usually possible to determine which component is having difficulty by looking at the run log file. You may then want to change the solution method for that component from among the options discussed below.

Solving method options fall into two categories: Component Solving Methods, and Matrix Solving Methods. Component Solving Methods refer either to modifications of the Newton-Raphson method, or a completely different method of finding values for the break variables that satisfy the component equations. Matrix Solving Methods refers to the way in which the next estimates of the break variables are determined from the current values using the Jacobian matrix.

Full explanation of the advanced methods is beyond the scope of this manual. The cited references were consulted in the *SPARK* implementation.

11.2 PREFERENCE SETTINGS

This section describes the preference settings used by *SPARK* to solve the components at runtime. Preference settings fall into two categories: global settings specific to all components and component settings specific to each component. The preference settings are specified in a preference file with extension .prf.

11.2.1 Default Preference File

The program *setupcpp* produces the solution sequence for the *SPARK* problem under study (See Figure 1-1). It also generates the probName.prf file that contains the list of preference settings for each component comprising the problem. The default settings written out to the probName.prf file come from hard-coded

[39] The solution sequence is derived by the *setupcpp* program from the topological information contained in the EVALUATE callback functions defined for each inverse, except for the SINK inverses that are automatically invoked last.

values unless a file named default.prf containing customized default values can be located in the current working directory.

The default.prf file defines the default global settings to be used within a segment starting with the `GlobalSettings` key. It also defines the default preference settings to be used for each strongly-connected component within a segment starting with the `DefaultComponentSettings` key. Note that no default settings need to be specified for component-specific entries such as the names of the trace files.

The following example shows a default.prf file with modified default values for the keys `ComponentSolvingMethod`, `MaxIterations` and `MatrixSolvingMethod`.

```
(
   GlobalSettings (
      Tolerance ( 1e-006 ())
      MaxTolerance ( 0.001 ())
      PredictionSafetyFactor ( 0.01 ())
      IterationSafetyFactor ( 0.9 ())
   ) // End of GlobalSettings section

   DefaultComponentSettings (
   // Settings for component solving method
      ComponentSolvingMethod (4 ())
      MinIterations ( 1 ())
      MaxIterations ( 1000 ())
   // Settings for Jacobian evaluation
      TrueJacobianEvalStep ( 0 ())
      JacobianRefreshRatio ( 0.1 ())
      Epsilon ( 0 ())
   // Settings for step control method
      StepControlMethod ( 1 ())
      RelaxationCoefficient ( 1 ())
      MinRelaxationCoefficient ( 1e-006 ())
   // Settings for matrix solving method
      MatrixSolvingMethod ( 4 ())
      ScalingMethod ( 0 ())
      PivotingMethod ( 1 ())
      RefinementMethod ( 0 ())
   ) // End of DefaultComponentSettings section
)
```

If no default.prf file resides in the current working directory, then *setupcpp* will generate a template default.prf file with the hard-coded default preferences for possible future modification by the user. Thus, specific default values for the component preference settings that differ from the hard-coded ones can be chosen for specific problems.

11.2.2 Global Settings

The global settings in the preference file are specified within a segment starting with the key `GlobalSettings`. They define the parameters used in the convergence check that must be satisfied by all components, such as the prescribed tolerance and various safety factors.

```
(
   GlobalSettings (
      Tolerance ( 1e-006 ())
      MaxTolerance ( 0.001 ())
      PredictionSafetyFactor ( 0.01 ())
```

```
      IterationSafetyFactor ( 0.9 ())
   ) // End of GlobalSettings section
   …
)
```

Table 11-1: Global preference settings.

Parameter [key in preference file]	**Allowed values**	**Notes**
Tolerance [Tolerance]	A floating point number>0.0	Solution relative tolerance. In iterative solution, iteration will continue until no variable y changes by more than *Tolerance**|*y*| between two successive iterations. Default = 1.E-6. See Section 11.6 for more details.
Maximum Tolerance [MaxTolerance]	A floating point number > Tolerance	Maximum Tolerance used for a "relaxed" tolerance check instead of Tolerance in case of no convergence after maximum iterations (see Tolerance definition above). Default = 1.E-3
Safety Factor for Break Unknowns [BreakUnknownSafetyFactor]	0 < floating point number	Safety factor applied to the convergence check for the break unknowns. Default = 1.
Safety Factor for Normal Unknowns [NormalUnknownSafetyFactor]	0 < floating point number	Safety factor applied to the convergence check for the normal unknowns. Default = 1.
Prediction Safety Factor [PredictionSafetyFactor]	0 < floating point number ≤ = 1.0	Safety factor applied during prediction convergence check. Default = 0.01 See Section 11.6 for more details.
Iteration Safety Factor [IterationSafetyFactor]	0 < floating point number ≤ = 1.0	Safety factor applied during iteration convergence check. Default = 0.9 See Section 11.6 for more details.

11.2.3 Default Component Settings

The default component settings are specified within a segment starting with the DefaultComponentSettings key. They define the default values for the settings that will be used when solving each component unless some settings are overloaded later in the preference file for the component in question. If no other settings are specified, then the default component settings will be used.

```
 (
   …
   DefaultComponentSettings (
      ComponentSolvingMethod ( 0 ())
      MinIterations ( 1 ())
      MaxIterations ( 50 ())
      …
   ) // End of GlobalSettings section
   …
)
```

The default component settings define the parameters used by the component solving methods (See Section 11.3), the matrix solving methods (See Section 11.4), and the Jacobian evaluation methods (See Section 11.5).

11.2.4 Component Settings

The component settings are specified within a segment starting with the `ComponentSettings` key. Then, the settings for each component are specified in a separate segment starting with the evaluation order of the component, zero indicating the first component.

Any of the default component settings can be overloaded for each individual component by specifying a new value for the key. For example, the following code snippet showing the portion of a preference file overloads the maximum number of iterations allowed in component 0, i.e., the first component of the solution sequence. The settings for the component 0 becomes 100, from the default value of 50 in Section 11.2.3.

```
(
   …
   ComponentSettings (
      0 (
         MaxIterations ( 100 ())
         …
      )
      …
   ) // End of GlobalSettings section
   …
)
```

11.2.5 Changing the Preference Settings

When the problem is executed the solving method settings and associated parameters are taken from the problem preference file probName.prf.

If you use *WinSPARK* or *VisualSPARK* graphical user interface, you can use provided menus for setting the solving methods and parameters, and the settings you specify will be transferred to the problem preference file.

You can also edit the problem preference file generated by *setupcpp* with any text editor. However, you have to be careful to respect the format of the preference file where an entry `ENTRY` for a key `KEY` is specified with the following syntax (See Appendix C):

```
   KEY  ( ENTRY ( ) )
```

If for any reason the preference file does not define a particular method or parameter, default settings built into the source code are used. These default settings are given in the tables below. These are “safe” but not necessarily recommended settings, so you should normally provide appropriate settings for your problem.

11.3 COMPONENT SOLVING METHODS

The available methods for solving the component are listed in Table 11-2. The code numbers are needed only if you want to set the option by editing the probName.prf file. To set the component solving method in the preference file, the `ComponentSolvingMethod` key must be set to the desired code number under the `ComponentSettings` key for the component in question. When using a graphical user interface the available choices are on a selection menu. Note that the solving method chosen will depend on the component. For example, non-iterative components do not need any solution method. You can examine the probName.eqs file to see how many break variables there are for each component.

Table 11-2: Component Solving Methods

Method	Code	Notes	Reference
Newton-Raphson	0	With or without relaxation (default).	(Conte and de Boor 1985)
Perturbed Newton	1	Solves a perturbed nonlinear model with Newton-Raphson. Very computationally expensive but effective with badly-conditioned systems.	(Dennis and Schnabel 1996)
Fixed point iteration	2	Successive substitution	
Secant	4	Multidimensional secant (using Broyden's update formula).	(Press, Flannery et al. 1988) (Dennis and Schnabel 1996)

In addition to the basic solution method for a component, there may be parameters that control how the method behaves. Available control parameters as shown in Table 11-3. For example, with Newton-Raphson method you may want to use relaxation, whereby the calculated corrections to the break variables are only partially applied. This is achieved by using a fractional relaxation coefficient. Additionally, in some cases it may be beneficial to scale the Jacobian matrix.

The default values in the table are used only if the parameter in question is not defined in the probName.prf file.

Table 11-3: Component Solution Parameters

Parameter [key in preference file]	Allowed values	Notes
Minimum Iterations [MinIterations]	An integer >=0	Minimum number of iterations to perform when iterative solution is used. Default = 1
Maximum Iterations [MaxIterations]	An integer >0	Maximum allowed iterations when iterative solution is used. Default = 50
Jacobian Evaluation Step [TrueJacobianEvalStep]	Integer >= 0	The Jacobian will be re-evaluated only after this number of iterations. Default = 0 (Automatic Jacobian evaluation). See Section 11.5.
Epsilon [Epsilon]	A floating point number >= 0.0	Change in independent variable used to evaluate the partial derivatives for Jacobian calculation. Default = 0 (see Section 11.5.1).
Step Control Method [StepControlMethod]	Integer >= 0	Controls the length of the step computed by the component solving method to achieve "global" convergence. 0 = (Default) Fixed relaxation; 1 = Backtracking, with basic halving strategy, attempting to decrease the scaled Euclidean norm of residuals; 2 = Backtracking with line search.[40] 3 = Affine invariant backtracking strategy.

[40] (Dennis and Schnabel 1996) should be consulted for more details on the backtracking step control algorithm with line search.

Relaxation Coefficient [RelaxationCoefficient]	0 < Floating point number <= 1.0	This is a multiplier applied to the Newton-Raphson calculated change to get the actual change during the iteration. • Fixed relaxation coefficient used with the step control method 0. • With the other step control strategies, this is the initial relaxation coefficient used to start the backtracking method. Default = 1.0
Minimum Relaxation Coefficient [MinRelaxationCoefficient]	0 < floating point number ≤ 1.0	Minimum relaxation allowed with the backtracking step control methods. Default = 10^{-6}
Scaling Method [ScalingMethod]	Integer >= 0	Scales the Jacobian before using it. Default = 0. 0 = No scaling; 1 = full affine invariant scaling (row and column scaling). See Section 11.7.3 for more details.

11.4 MATRIX SOLVING METHODS

In Newton-Raphson and related component solving methods a linear set of equations must be solved at each iteration (see Section 2.5), yielding a correction to the current estimate of the cut set variables. By default, *SPARK* will use Gaussian elimination to effect this solution. However, other options are available as shown in Table 11-4. The code numbers are needed only if you want to set the option by editing the probName.prf file. To set the matrix solving method in the preference file, the `MatrixSolvingMethod` key must be set to the desired code number under the `ComponentSettings` key for the component in question.

Table 11-4: Matrix Solving Methods

Method	Code	Notes	Reference
Gaussian Elimination	0	Default	(Conte and de Boor 1985)
Singular Value Decomposition (SVD)	1	Poorly conditioned matrix.	(Press, Flannery et al. 1988)
Lower-Upper Factorization (LU)	2		(Conte and de Boor 1985)
Sparse LU	4	Sparse Matrix	http://www.cise.ufl.edu/research/sparse/umfpack/

The sparse linear solution method is selected by specifying the value 4 for the key `MatrixSolvingMethod` in the problem.prf file. This solution method uses the *C* library *UMFPACK* 4.0 developed by Tim Davis. The library implements the LU solution technique with column reordering for sparse linear systems. The linear solver does not rely on vendor-specific BLAS routines but instead on vanilla *C* code, thus ensuring portability of the *SPARK* program. Gains in calculation speed by many orders of magnitude have been observed on large problems for which the Jacobian matrix is typically more than 90% sparse.

Table 11-5: Matrix Solving Method Parameters

Parameter [`key` **in preference file]**	**Values**	**Notes**
Pivoting Method [`PivotingMethod`]	0, 1, 2	Only used with the Gaussian Elimination matrix solving method. 0 = No pivoting; 1 = (default) Partial pivoting, row pivots; 2 = Total pivoting, rows and columns.[41]
Refinement Method [`RefinementMethod`]	0 < Integer < 5	Only used with the LU solving matrix solving method. Indicates the number of refinement iterations.

11.5 JACOBIAN EVALUATION METHODS

The following keys specified in the problem preference file allow you to control the evaluation methods for the Jacobian matrix required by the various Newton-based iterative solution methods.

11.5.1 Scaled Perturbation for the Numerical Approximation of the Partial Derivatives

In *SPARK*, Newton-based iterative solution methods (i.e., Newton-Raphson) require the Jacobian matrix to be computed. This matrix consists of the partial derivatives of the iterated system of equations with respect to the break variables. These partial derivatives are approximated by **forward finite differences**.

For example, the partial derivative of the equation $f(t, x, y)$ with respect to the break variable y is approximated using the following formula:

$$\frac{\partial f(t,x,y)}{\partial y} \approx \frac{f(t,x,y+\Delta y_{FD}) - f(t,x,y)}{\Delta y_{FD}} \tag{11.1}$$

Here, Δy_{FD} is called the **perturbation** value of the variable y. You can specify the value of the perturbation value for each component using the keyword `Epsilon` in the problem preference file (see Section 11.2.2).

The differencing procedure in digital computation is sensitive to roundoff error. The main source of difficulty in computing the Jacobian matrix by finite differencing is the choice of the perturbation Δy. Consequently, *SPARK* provides the option to use a **scaled perturbation** value to compute the partial derivatives. This is done by specifying a 0 value for the `Epsilon` component setting in the preference file for the component in question.

For example, if you wish to use scaled perturbation in Component 0, the preference file should include:

```
ComponentSettings (
   0  (
      Epsilon ( 0 ())
      …
      )
)
```

When `Epsilon` is specified as zero, *SPARK* computes the scaled perturbation value for the variable y as:

[41] The Gaussian elimination solving method with full pivoting is also referred to as the Gauss-Jordan elimination solving method in (Press, Flannery et al. 1988).

$$\Delta y_{FD} = sign(\dot{y}) \cdot \max\left(|y|, |y + h \cdot \dot{y}|, atol(y) \right) \cdot \sqrt{URound} \tag{11.2}$$

Here, $URound$ is the machine unit round-off error. The derivative, $\dot{y}$, with respect to the independent variable (usually time) is approximated using the explicit Euler scheme. The term $|y + h \cdot \dot{y}|$ is included to represent the predicted value for *y* at the next step. This is because even if $|y|$ happens to be near zero, it is quite possible that a nearby value of *y* is not so small, and selecting $|y + h \cdot \dot{y}|$ will prevent a near zero perturbation from being used. In the event that $|y|$ and $|y + h \cdot \dot{y}|$ are both near zero, the absolute error tolerance $atol(y)$ is used as a lower bound in the formula to prevent using too small a perturbation. Indeed, by setting the error tolerance, you tell the *SPARK* solver that it is the smallest number which is relevant with respect to the break variables *y* in this component.

The formula in Equation (11.2) perturbs about half of the digits of the variable *y* when *y* is significantly larger than $atol(y)$. Finally, note that the sign of the perturbation Δy_{FD} computed with Equation (11.2) will be negative if the solution is decreasing. Unfortunately, this choice is a potentially source of difficulty for problems where some functions are undefined for $y < 0$ or not differentiable at $y = 0$.

11.5.2 Jacobian Refresh Strategy

The key `TrueJacobianEvalStep` in the problem preference file specifies the iteration frequency at which the Jacobian matrix is evaluated. For example, setting

```
TrueJacobianEvalStep  ( 1 ( ))
```

indicates that the Jacobian matrix for the strongly-connected component in question will be evaluated at each iteration.

For example, setting the value to 5 indicates that the Jacobian matrix will be refreshed after 5 iterations, starting at the first iteration of each new time step.

11.5.3 Automatic Jacobian Refresh Strategy

Refreshing the Jacobian matrix is a costly operation that requires firing the system of equations as many times as there are break variables in the strongly-connected component. Therefore, an efficient solver should try to minimize the number of times the Jacobian matrix needs to be refreshed in order to still achieve fast convergence of the solution methods.

By setting in the problem preference file

```
TrueJacobianEvalStep  ( 0 ( ))
```

the Jacobian will be refreshed automatically and "optimally" by the *SPARK* solver whenever it is needed to ensure satisfactory convergence.

By default, the *SPARK* solver uses the automatic Jacobian refresh strategy unless specified otherwise in the problem preference file.

The automatic refresh strategy is based on the convergence behavior of the scaled increment norms between successive iterations. The Jacobian matrix is refreshed whenever the convergence rate $\Theta^{(k+1)}$ becomes greater than the user-specified Jacobian refresh ratio $\Theta_{JacRefresh}$:

$$\Theta^{(k+1)} = \frac{\left\| D_{x^{(k)}}{}^{-1} \cdot \Delta x^{(k+1)} \right\|}{\left\| D_{x^{(k)}}{}^{-1} \cdot \Delta x^{(k)} \right\|} > \Theta_{JacRefresh} \tag{11.3}$$

The value of the Jacobian refresh ratio is set by default to 0.5 in the *SPARK* solver. Thus, we request that the increment norms be at least halved between successive iterations otherwise the Jacobian matrix is refreshed at the next iteration.

The value of the Jacobian refresh ratio can be changed by specifying a positive floating-point value less than or equal to one with the key `JacobianRefreshRatio` in the problem preference file. For example, setting in the problem preference file

```
JacobianRefreshRatio  ( 0.01 ( ))
```

forces the Jacobian matrix to be refreshed at every iteration for which the increment norm has not been decreased by at least two orders of magnitude since the previous iteration.

11.6 CONVERGENCE CHECK STRATEGY

11.6.1 Notation

We introduce the notation $x^{(k)}$ to refer to the values of the vector *x* at the iteration *k*.

The notation $x_i^{(k)}$ refers to the *i*-th element of the vector *x* at the iteration *k*.

However, a superscript not enclosed within brackets refers to a normal power applied to the variable, e.g. x^2 refers to the square of the vector *x*.

The notation D_{ij} refers to the element in row *i* and column *j* of the matrix *D*.

Finally, the notation $\|x\|$ refers to the norm of the vector *x*.

11.6.2 Scaled Stopping Criterion for Iterative Solution

Consider the natural stopping criterion for a Newton method in its unscaled form. That is, at iteration (*k+1*), for the vector *x* of the unknown variables, the convergence criterion requires that the iteration error *err* be smaller than the prescribed tolerance tol. The iteration error $err = x^* - x^{(k)}$ is reasonably estimated by the iteration increment $\Delta x^{(k)} = x^{(k+1)} - x^{(k)}$ since the Newton method converges quadratically near the solution x^*.

$$\begin{aligned} & err \simeq \left\| \Delta x^{(k)} \right\| = \left\| x^{(k+1)} - x^{(k)} \right\| \\ & stop, if\ err \le tol \\ & tol : prescribed\ (required)\ tolerance\ (accuracy) \end{aligned} \tag{11.4}$$

In this unscaled form, *err* is a measure of the **absolute error** of the numerical solution $x^{(k+1)}$.

Note that $\Delta x^{(k)}$ is the true iteration increment that factors in the effect of any relaxation coefficient. Thus, if we apply a relaxation coefficient $0 < \lambda^{(k)} \le 1$ at the current iteration (*k*), the iteration increment used in the convergence check would be related to the Newton step $\Delta x_{Newton}^{(k)}$ through :

$$\Delta x^{(k)} = \lambda^{(k)} \cdot \Delta x_{Newton}^{(k)} = -\lambda^{(k)} \cdot J\left(x^{(k)}\right)^{-1} \cdot F\left(x^{(k)}\right) \tag{11.5}$$

A proper internal scaling procedure plays an important role for the efficiency and robustness of any algorithm. A desirable property of an algorithm is the so-called **scaling invariance**. This means, e.g., rescaling of some or all components of the vector of unknowns, *x*, (say, from mm to km) should not affect the algorithmic performance, although the problem formulation may change.

SPARK employs a scaled tolerance test as the stopping criterion used to decide when to terminate the iterative solution of a strongly connected component. The scaled tolerance test for the problem variable x_i, at the iteration (*k+1*), is

$$err\left(x_i^{(k+1)}\right) \simeq \left|\frac{x_i^{(k+1)} - x_i^{(k)}}{scale\left(x_i^{(k)}\right)}\right| \leq tol \tag{11.6}$$

where $scale\left(x_i^{(k)}\right)$ is the scale for the variable x_i based on the value at the iteration (*k*). In this scaled form, $err\left(x_i^{(k+1)}\right)$ becomes a measure of the **relative error** in $x_i^{(k+1)}$.

The value of the relative tolerance tol is specified with the key `Tolerance` in the problem preference file for each strongly connected component.

It is recommended to use the same value of the relative tolerance for all strongly connected components to ensure that the global relative tolerance achieved in the solution of the entire problem is consistent. If one component is solved with a larger relative tolerance then the accuracy achieved in all components downstream will be limited by this larger value no matter what their individual, possibly stricter, relative tolerance is.

11.6.3 Prediction Convergence Check

Before the first iteration, the residual function $F\left(x^{(0)}\right)$ is evaluated with the predicted values for the break variables $x^{(0)}$. If the following condition

$$err\left(F\left(x_i^{(0)}\right)\right) \simeq \left|\frac{F\left(x_i^{(0)}\right)}{scale\left(x_i^{(0)}\right)}\right| \leq safety_{pred} \cdot tol,\ i = 1 \ldots n \tag{11.7}$$

holds for the residual function associated with the break variables, then the predicted solution $x^{(0)}$ is accepted as the converged solution without proceeding with any further iteration. Because the convergence test occurs in the residual space, the tolerance test is typically made stricter by multiplying with $safety_{pred} \leq 1$. By default, $safety_{pred} = 0.01$ in the solver.

This test is intended to avoid iterating when the predicted state of the underlying system is already very close to the solution because:

- the prediction is very good (e.g. when restarting from a snapshot file), or
- the dynamic problem has almost the same solution as at the previous time step (e.g. steady state solution).

This prediction convergence check has two main disadvantages:

- the non-break variables are not checked against their individual scaled tolerance, which can lead to a loss of accuracy (depending on the requested tolerance on the non-break variables and on the transfer function from the break variables to the non-break variables); and
- **hidden residual inverses** of the form $x_i = x_i + residual(\ldots, x_i, \ldots)$ may be scaled improperly with $scale\left(x_i^{(0)}\right)$ if $\left|\frac{\partial residual(\ldots, x_i, \ldots)}{\partial x_i}\right| \ll 1$, which in turn might lead to undesirable early convergence during the prediction.

To avoid these situations, it is possible to set the prediction safety factor $safety_{pred}$ in the problem preference file using the key `PredictionSafetyFactor`. For example, setting in the problem preference file

```
PredictionSafetyFactor ( 0 ())
```

ensures that the convergence check will never be satisfied during prediction unless the residual function $F\left(x^{(0)}\right)$ is exactly null.

11.6.4 Iteration Convergence Check

The solution at the iteration (k+1) in a strongly-connected component is accepted if the following conditions

$$\begin{aligned} &a) \qquad err\left(x_i^{(k+1)}\right) \simeq \left|\frac{x_i^{(k+1)} - x_i^{(k)}}{scale\left(x_i^{(k)}\right)}\right| \le \lambda^{(k)} \cdot safety_x \cdot tol,\ i = 1 \ldots n \\ &b) \qquad err\left(y_j^{(k+1)}\right) \simeq \left|\frac{y_j^{(k+1)} - y_j^{(k)}}{scale\left(y_j^{(k)}\right)}\right| \le safety_y \cdot tol,\ j = 1 \ldots m \end{aligned} \tag{11.8}$$

hold for each break variable x_i and each non-break variable y_j, where $\lambda^{(k)}$ is the relaxation coefficient for the current iteration.

Factoring in the relaxation coefficient in the convergence test *a*) for the break variables ensures that convergence will not be wrongly detected due to the application of a small relaxation coefficient when updating the iterate (k+1).

11.6.5 Safety Factors

The iteration safety factors appearing in Equation (11.8) for the break unknowns *x* and the normal unknowns *y* can be set using the safety factor parameters defined in the problem preference file. The safety factors $safety_x$ and $safety_y$ are computed from the preference parameters with the following equations:

$$\begin{aligned} &a) \qquad safety_x = BreakUnknownSafetyFactor \cdot IterationSafetyFactor \\ &b) \qquad safety_y = NormalUnknownSafetyFactor \cdot IterationSafetyFactor \end{aligned} \tag{11.9}$$

The iteration safety factor *IterationSafetyFactor* is set to 0.9 by default. It can be changed using the key `IterationSafetyFactor` in the problem preference file .

The two other parameters *BreakUnknownSafetyFactor* and *NormalUnknownSafetyFactor* let you control the convergence check for each type of unknowns. Default values for these parameters are 1, as by default *SPARK* applies the same safety factor in the convergence check for all unknowns. However, if you want to

relax the convergence check performed on the normal unknowns, you can simply set the entry for the key `NormalUnknownSafetyFactor` to a value bigger than 1. Similarly, the safety factor for the break unknowns can also be changed using the key `BreakUnknownSafetyFactor` in the preference file.

11.6.6 Relaxed Convergence Check

If convergence is not achieved after the maximum number of iterations specified with the key `MaxIterations` in the problem preference file, then *SPARK* performs a relaxed convergence check. The relaxed convergence check consists of using the maximum relative tolerance specified with the key `MaxTolerance` in the problem preference file in place of the normal relative tolerance specified with the key `Tolerance`.

$$err\left(x_i^{(k+1)}\right) \simeq \left|\frac{x_i^{(k+1)} - x_i^{(k)}}{scale\left(x_i^{(k)}\right)}\right| \leq \lambda^{(k)} \cdot safety \cdot tol_{relaxed}, i = 1 \ldots n \tag{11.10}$$

where $tol_{relaxed}$ is set to the value of `MaxTolerance`.

The scales $scale\left(x_i^{(k)}\right)$ are **not** re-computed to reflect the new relative tolerance requirement based on $tol_{relaxed}$. Thus, the relaxed convergence check also relaxes the accuracy requirement with respect to the absolute tolerance for each problem variable.

If $tol_{relaxed} > tol$, then the number of significant digits achieved in the solution for the variables not too near to their respective absolute tolerance specifications will be reduced from $-\log_{10}\left(tol\right)$ to $-\log_{10}\left(tol_{relaxed}\right)$.

The relaxed convergence test is based on the break variables **only** as opposed to the iteration convergence check, which also enforces the convergence check on the non-break variables.

If the previous condition for the relaxed convergence test holds, then *SPARK* writes a warning to the error log file and proceeds with the simulation; otherwise *SPARK* terminates with a convergence error message. The relaxed convergence check mechanism lets you carry out a dynamic simulation over multiple time steps until the final time even though a few time steps might not have been computed with the full desired accuracy.

The relaxed convergence check can be turned off by setting `MaxTolerance` to a value smaller or equal to `Tolerance` in the problem preference file.

11.7 SCALING METHODS

11.7.1 Variable Scaling Procedure

To achieve scaling invariance in the error estimation and to avoid the difficulties arising from near-zero problem variables the following scaling strategy is applied:

- initial update

$$typ\left(x_i^{(0)}\right) = \left|x_i^{(0)}\right| \tag{11.11}$$

- iteration update (taking into account two successive iterates)

$$typ\left(x_i^{(k)}\right) = \frac{1}{2}\left(\left|x_i^{(k)}\right| + \left|x_i^{(k-1)}\right|\right) \tag{11.12}$$

- scaling procedure

$$scale\left(x_i^{(k)}\right) = \max\left\{typ\left(x_i^{(k)}\right), threshold\left(x_i^{(k)}\right)\right\} \tag{11.13}$$

where the threshold value $threshold\left(x_i^{(k)}\right)$ for scaling must be strictly positive and is specific to each variable x_i.

Note that the actual value of $threshold\left(x_i^{(k)}\right)$ determines a switch from a pure relative norm to a modified absolute norm for each problem variable.

As long as $\left|x_i^{(k)}\right| > threshold\left(x_i^{(k)}\right)$, this problem variable contributes

$$\frac{\Delta x_i^{(k)}}{typ\left(x_i^{(k)}\right)}$$

to the norm, whereas for $\left|x_i^{(k)}\right| \le threshold\left(x_i^{(k)}\right)$ this problem variable contributes

$$\frac{\Delta x_i^{(k)}}{threshold\left(x_i^{(k)}\right)}$$

to the norm.

Defining the Absolute Tolerance for Each Problem Variable with the ATOL Property

The threshold value $threshold\left(x_i\right)$ is specific to each problem variable and is derived from the absolute tolerance value $atol\left(x_i\right)$ specified for each unknown variable with the `ATOL` keyword in the `LINK`, `PORT` or `PROBE` statements.

$$threshold\left(x_i\right) = \frac{atol\left(x_i\right)}{tol} \tag{11.14}$$

The `ATOL` property should be set to the absolute value at which the variable in question is essentially insignificant, i.e. it is no longer necessary to request further accuracy in terms of significant digits for values smaller than the absolute tolerance. By default, the `ATOL` property is set to 10^{-6} if it is not explicitly specified for a variable.

For example, the following statement in a *SPARK* class file or problem file indicates that the absolute tolerance for the variable `massFlow` is to be set to 10^{-10} :

```
LINK massFlow   o1.m  io2.m  INIT=0   ATOL=1.0E-10  [kg/s];
```

Of course the value of the absolute tolerance depends on the physical units used in the problem formulation. However, changing the requested relative tolerance with which to solve the problem does not impact the choice of the absolute tolerance since the threshold value is automatically adjusted to reflect the new prescribed relative tolerance.

Achieved Accuracy

Such a scaled tolerance requirement is necessary to achieve convergence with a consistent number of significant digits, *p*, for variables with different orders of magnitude.

For a problem variable $|x_i| > threshold(x_i)$, the relationship between the relative tolerance and the number of significant digits, *p* (indicating the number of correct decimal leading digits in the mantissa of each problem variable x_i independent of the actual exponent) achieved in the solution is:

$$p = -\log_{10}(tol) \tag{11.15}$$

For a problem variable $|x_i| \le threshold(x_i)$, the number of correct digits *p* in the mantissa is:

$$p = -\left[\log_{10}(tol) - \left[\log_{10}(x_i) - \log_{10}(scale(x_i))\right]\right] \tag{11.16}$$

In other words, the **absolute error** for each problem variable, in both cases, is approximately given by:

$$abs\_err(x_i) \approx tol \cdot scale(x_i) \tag{11.17}$$

11.7.2 Scaled Norms and Implications for the Solution Methods

In *SPARK*, we use scaled norms in place of the usual unscaled norm in order to obtain norms that are scaling invariant. The scaling matrix and norm used in the code are given by:

$$\begin{aligned} &a) \quad D = diag(scale_1, \ldots, scale_n) \\ &b) \quad \|v\|_{scaled} = \left\|D^{-1} \cdot v\right\| = \sqrt{\sum_{i=1}^{n}\left(\frac{v_i}{D_{ii}}\right)^2} \end{aligned} \tag{11.18}$$

In *SPARK* all vector norms $\|v\|$ are Euclidean norms (a.k.a. 2-norms) unless specified differently.

In the Variable Space

The described scaling procedure yields reasonable values for the scaled norms used in *SPARK* in the variable space. For example, when we report the variable increments norm it is assumed that it is the scaled norm of the increments that is computed:

$$\|\Delta x\|_{scaled} = \left\|D_x^{-1} \cdot \Delta x\right\| \tag{11.19}$$

where the scaling matrix D_x is the diagonal matrix with the scales $scale(x_i)$ for each problem variable x_i (See Equation (11.13)).

In the Residual Space

However, norms computed in the residual space tend to be more difficult to make scaling invariant. An unscaled termination criterion in the space of the residuals

$$\|F(x)\| \le tol \tag{11.20}$$

neither controls the error in the computed solution nor shows any invariance property. In order to realize invariance against a rescaling of the residuals, one may use a scaled check, e.g.

$$\left\| F\left(x\right)\right\|_{scaled} = \left\| D_F^{-1} \cdot F\left(x\right)\right\| \le tol \tag{11.21}$$

where

$$D_F = diag\left(typ\left(F_1\left(x\right)\right),\ldots,type\left(F_n\left(x\right)\right)\right) \text{ with } D_{F\,ii} > 0, i = 1\ldots n \tag{11.22}$$

However, the selection of the typical values $typ\left(F_i\left(x\right)\right)$ for the residuals is arbitrary. Furthermore, it is not obvious how to develop an adaptive selection of further scaling matrices when the residual function $F\left(x\right)$ evolves over successive iterations.

To avoid such a situation, *SPARK* checks for convergence in the variable space after the prediction step (see Section 11.6.4).

11.7.3 Total Internal Scaling of Linear Systems

All component solving methods in *SPARK* except for the fixed point iteration require solving a linear system of the following general form in order to compute the Newton step:

$$J \cdot \Delta x = -F\left(x\right) \tag{11.23}$$

This is solved for the increment vector Δx using the vector of residuals $F\left(x\right)$ and the Jacobian matrix J that contains the partial derivatives of the residuals with respect to the vector of break variables x.

In order to have scaling invariance for the linear system solution, the associated linear systems can be internally scaled by setting the scaling method to 1 in the problem preference file.

```
ScalingMethod  ( 1 ( ))
```

The scaling method in *SPARK* implements a fully affine invariant scaling scheme in both the variable space and the residual space by applying column scaling and row scaling to the linear system. This scaling approach makes the solver operation less sensitive to changes in the variables' units and to equation formulations where the variables show very different orders of magnitude. In particular, a user rescaling of the dependent variables does not change the performance of the linear solver.

The total internal scaling consists in solving the following row- and column-scaled linear system:

$$\left(D_F^{-1} \cdot J \cdot D_x\right) \cdot \left(D_x^{-1} \cdot \Delta x\right) = -D_F^{-1} \cdot F\left(x\right) \tag{11.24}$$

Herein, D_x is the diagonal matrix with the scales $scale\left(x_i\right)$ of the break variables, updated at each iteration using the scaling scheme described in Section 11.7.1.

$$D_x = diag\left(scale\left(x_1\right),\ldots,scale\left(x_n\right)\right) \tag{11.25}$$

D_F is another diagonal matrix.

$$D_F = diag\left(d_1,\ldots,d_n\right) \tag{11.26}$$

Let $\left(J \cdot D_x\right)_{ij}$ denote the elements of the column scaled Jacobian $J \cdot D_x$. Then the residual scale d_i is calculated according to

$$d_i = \max_{1 \le j \le n} \left| \left(J \cdot D_x \right)_{ij} \right|, \quad i = 1, \ldots, n \tag{11.27}$$

If you encounter convergence difficulties with the solution of a *SPARK* problem, the fully affine invariant scaling scheme should be selected in the problem preference file to improve the operation of the nonlinear solver.

By default, the *SPARK* solver does not perform the total internal scaling of the linear systems in order to avoid the associated performance penalty.

11.7.4 Detection of an Ill-Conditioned Problem

Assume that the nonlinear problem

$$\begin{aligned} &a) \quad F\left(x\right) = 0,\ x \in \mathbb{R}^n \\ &b) \quad x^{(0)}\ \textit{given initial guess} \end{aligned} \tag{11.28}$$

is well-scaled, i.e., unscaled norms yield meaningful numbers. If the situation

$$\left\| \Delta x \right\| \le tol \text{ with } \left\| F\left(x\right) \right\| \text{ "large"} \tag{11.29}$$

holds, the underlying problem is said to be **ill-conditioned**. That means that a large value for $\left\| F\left(x\right) \right\|$ may occur even for $x = float\left(x^*\right)$ since x^* can't be represented exactly due to the finite length of the mantissa.

For a badly-scaled problem, a check for the condition of the problem must rely on scaled norms. The following situation

$$\left\| D_x^{-1} \cdot \Delta x \right\| \le tol \text{ with } \left\| D_F^{-1} \cdot F\left(x\right) \right\| \text{ "large"} \tag{11.30}$$

indicates an ill-conditioned problem, provided that the scaling matrices are properly chosen. Ill-conditioned problems are numerically difficult to solve because the achievable precision might be limited.

In the case of a non-converging, ill-conditioned problem, you should consider relaxing the tolerance requirement (the relative tolerance and the absolute tolerances for the worst offending variables) in order to obtain a trustworthy solution albeit with less accuracy.

11.7.5 Implication for the Backtracking Step Control Methods

The step control methods based on the backtracking approach (i.e., the basic halving strategy and the line search strategy) aim at minimizing a cost function *f* based on the norm of the residuals by adapting the relaxation coefficient to be applied at each iteration.

$$\min_{x \in \mathbb{R}^n} f\left(x\right) = \frac{1}{2} \left\| F\left(x\right) \right\|^2 \tag{11.31}$$

It is clear that if the units and/or orders of magnitude of two components of the residual function $F\left(x\right)$ are widely different, then the smaller component function will be virtually ignored by not contributing much to the norm of the residual function.

For this reason, the backtracking algorithms in *SPARK* use a positive diagonal matrix $\bar{D}$ on the dependent variables $F(x)$. The diagonal matrix is chosen so that all the components of $F(x)$ will have about the same typical magnitude at points not too near the root. Thus, the cost function defined in *SPARK* is

$$\min_{x \in \mathbb{R}^n} f(x) = \frac{1}{2} \left\| \bar{D}^{-1} \cdot F(x) \right\|^2 \tag{11.32}$$

The residual functions $F_i(x)$ are derived from the directed inverses $inverse_i(x)$ assigned to each break variable x_i in the atomic classes:

$$F_i(x) = inverse_i(x) - x_i, \; i = 1 \ldots n \tag{11.33}$$

Because each residual function $F_i(x)$ depends on the break variable x_i, the default choice in *SPARK* for the matrix $\bar{D}$ is the variable scaling matrix D_x.

However, if the linear system is scaled using the total internal scaling scheme (see Section 11.7.3), then *SPARK* uses the internally computed, row scaling matrix D_F in place of the matrix $\bar{D}$ to compute the cost function. This is a better choice as it takes into account the dependency on all *x*'s and not just on x_i for each inverse $inverse_i(x)$.

Thus, selecting the total internal scaling scheme impacts the operation of the nonlinear solver by modifying the cost function used with the backtracking step control methods. Therefore, if a problem fails to converge with the scaling method turned off, convergence can sometimes be achieved when re-computing the same step with the scaling method turned on (and vice versa).

12 DEBUGGING *SPARK* PROGRAMS

Often *SPARK* will find calculation sequences leading to successful problem solution without intervention. However, solution of nonlinear differential and algebraic equations is not easy, even for *SPARK*, and in some cases you may get error messages. These may be during the initial processing where your input is being parsed, while executing the setup program that converts it to a solver program, or during execution of the solver program, i.e., at run time.

12.1 PARSING ERRORS

Parsing errors are usually syntax errors, as in any programming language. These errors are reported in the parser.log file, normally placed in your project directory. They should be easy to interpret, but if not the command reference in Section 19 may be helpful.

12.2 SETUP ERRORS

During the setup phase *SPARK* may have other difficulties due to input errors. For example, you may have specified a problem for which no matching can be found between equations and variables. This can happen even if you have an equal number of equations and free variables (i.e., links). As an example of this, consider the **4sum** problem when *x1, x5, x6,* and *x7* are specified as inputs. This is not well-posed because it over-determines the equation for *s3* while under-determining *s2*. *SPARK* will report such errors as "unable to find a matching". Subtle errors of this nature can occur in development of complex models. Setup errors are reported in the setup.log file.

Unfortunately, lack of matching can also arise for well-posed problems if you have not provided enough inverses for your atomic objects. Complex models involve equations that maybe difficult to invert, even with symbolic algebra tools. Consequently, it is common for *SPARK* users to omit the difficult inverses for some equations, providing only those easily come by. Usually, this is acceptable practice since *SPARK* explores many paths to a get a solution sequence and usually finds one.

However, if you are experiencing matching problems and have omitted some inverses you may want to consider using **residual inverses** (See Section 8.7) or **default residual inverses** (See Section 8.8) to facilitate the matching process.

12.3 SOLUTION DIFFICULTIES

Even after *SPARK* has successfully created a solver program there can be difficulties in finding a solution. This is because of the nature of nonlinear systems of equations, with which numerical analysts have been struggling for many years. Here we are referring to convergence difficulties; the solver iterates the maximum allowed number of times (set by default to 50) without bringing the solution into the error tolerance (default value is 10^{-6}). If you work with complex systems, resolving these difficulties is the greatest challenge you will face. Run time errors are reported to the run log file. More detailed error messages and diagnostic can be found in the error log file (see Section 16.1).

With *SPARK*, you attack convergence problems in two basic ways: estimating better values to start the iteration, and by trying to alter the solution sequence. The importance of good iteration initial values is well known; in this regard, the only difference between *SPARK* and other simulation tools is with *SPARK*, due to reduction in the number of iteration variables, you do not have to specify as many guess values. We discuss how to set initial iteration values in Section 7.2.

The second strategy, controlling the solution sequence, is based on the observation that iteration can usually be done many different ways, often differing in the direction in which calculations flow around cycles in the

problem graph. Sometimes convergence can be achieved by calculating in the opposite direction. Consequently, *SPARK* provides syntax in the definition of problems and classes in order to control, indirectly, the calculation direction. You can always see the solution sequence chosen by *SPARK* in the .eqs file produced by the setup program. Open this file with a suitable viewer or editor and use it as guide in understanding and improving your problem solution sequence.

MATCH_LEVEL is very effective in reversing the direction of calculations in *SPARK*. By default, matchings are found based only on order of objects and links found in the problem specification file. By forcing or encouraging a different matching you can often improve numerical performance, and perhaps achieve convergence.

The relevant keywords are MATCH_LEVEL and BREAK_LEVEL. Each can be set to a value between 0 and 10. When left unspecified, these levels default to 5. The MATCH_LEVEL keyword is placed in a LINK or PORT statement, and specifies the relative desirability of matching that link variable to a particular object in the LINK statement. For example,

```
LINK  x  a_obj.p1 MATCH_LEVEL = 10, b_obj.p3;
```

tells *SPARK* that you would prefer that object *a_obj* should be matched with the *x* problem variable. You could say somewhat the same thing by the statement

```
LINK  x  a_obj.p1, b_obj.p3 MATCH_LEVEL = 0;
```

which says you would prefer that *x* not be matched with object *b_obj*. Provided that you not simply encourage selection of the matching that would be found by default, the direction of calculations in the problem will be reversed. Currently, the second form is stronger that the first due to the implementation of the matching algorithm used in *SPARK*.

BREAK_LEVEL parallels the MATCH_LEVEL idea, but applies to the discovery of a cut set, i.e., selection of variables to break cycles in the problem graph. When there is a cycle, usually many problem variables are encountered as you work your way around the loop. It is easy to see that any of these variables will break the loop. By default, *SPARK* sets break preference to 5 for all variables, so the break selected is determined solely by order in the problem definition. Yet, there are sometimes arguments for preferring one over another.

A simple example is based on starting value availability. If you have the choice of breaking on enthalpy or temperature, you may prefer the latter simply because you are likely to be able to better estimate iteration starting values for temperature. Some analysts also feel that different break variables lead to better convergence. However, the “gain” around the loop is going to be the same regardless, so this may not be a strong argument. Nonetheless, if you have any reason or hunch that a particular variable would be a better break, give it a high BREAK_LEVEL. To do so, include it in the LINK statement:

```
LINK  x a_obj.p1 BREAK_LEVEL = 7, b_obj.p3 MATCH_LEVEL = 10;
```

In the current implementation, matching and break levels only encourage *SPARK* to match or break the way you wish. This is because we wanted to give *SPARK* maximum opportunity to find solution sequences, and denying certain matchings and breaks may prevent any solution at all. In later versions we may also provide forced matchings and breaks.

Finally, it should be noted that these are only indirect tools, sometimes having little or no effect on the solution sequence. For example, setting BREAK_LEVEL on a link that does not happen to be in a cycle will have no effect, and as already noted setting a MATCH_LEVEL to force a match that is selected by default is also ineffective.

12.4 TRACE FILE MECHANISM

Sometimes it may be helpful to see intermediate results of the iterative solution process. This is especially important when your problem is experiencing convergence difficulties. You can get such output by using the `TraceFiles` segment under the key `ComponentSettings` for the component in question in the probName.prf file. This is done for individual components.

As with solution control parameters (see Section 11), setting this flag is done most conveniently with the aid of a *SPARK* graphical user interface. Otherwise, you can edit the probName.prf file directly with any text editor.

The `TraceFiles` segment has five allowed values as shown in Table 12-1.

Table 12-1: Keys and Values for the `TraceFiles` Segment

`TraceFiles` **Key and Value**	**Meaning**
`( )`	No trace output.
`Jacobian (fileName () )`	Jacobian of residual functions printed whenever it is recomputed.
`Increments (fileName () )`	Increments of all variables printed at every iteration.
`Residuals (fileName () )`	Break residuals printed at every iteration.
`Variables (fileName () )`	All problem variables printed at every iteration.

Within each component, you can specify up to four trace files entries with the name of each file preceded by one of the keys listed in Table 12-1. Each key specifies the type of the trace file that will be written to the file following the type key. For example, the following segment could be inserted in `ComponentSettings` 0 of a problem preference file:

```
ComponentSettings (
   0  (
         …
         TraceFiles (
            Jacobian   ( spring_jac.trc ())
            Increments ( spring_inc.trc ())
            Residuals  ( spring_res.trc ())
            Variables  ( spring_var.trc ())
         )
         …
   )
)
```

Any file name with the extension .trc can be used, except it cannot be repeated. That is, you cannot use the same file name for tracing in the same component, or in a different component.

If no trace files are wanted, the `TraceFiles` segment for the component should be:

```
   TraceFiles ( )
```

Finally, note that only the variable tracing option is available with weak components.

12.5 PROBLEM-LEVEL DIAGNOSTIC MECHANISM

In addition to the trace facility (see Section 12.4), *SPARK* has a problem-level diagnostic facility. To use this feature, the `DiagnosticLevel` keyword must be set to something higher than 0 in the problem run-control file (see Section 18). The different modes trigger increasing level of diagnostic to the run log file.

The default mode is the silent mode. To combine diagnostic modes, you add the corresponding flag values and specify the resulting value for the `DiagnosticLevel` key. For example, to produce diagnostic about the input mechanism, the report mechanism and the simulation statistics, the value 2+4+16=22 should be specified for the `DiagnosticLevel` key.

Table 12-2: Problem-level Diagnostic Flag Values

Mode	Flag value	Description
Silent	`0`	No diagnostic. Default mode if no diagnostic level is specified.
Show convergence	`1`	At each iteration, the convergence progress is reported for each component. Includes scaled residuals' norm, convergence error, requested tolerance, name and value of the worst-offender variable.
Show inputs	`2`	All variables read from input files or URL are listed before the beginning of the simulation.
Show reports	`4`	All variables tagged as `REPORT` are reported with their names and values at each step.
Show preference settings	`8`	Loaded preference settings are written out before the beginning of the simulation.
Show simulation statistics	`16`	Simulation statistics is produced at the end of the simulation.
Show requests	`32`	Internal and external requests are traced over the course of the simulation.

12.5.1 Description of the Inputs Diagnostic Mode

When the inputs diagnostic mode is selected, the variables for which values are specified in input files or through Read URLs are written to the run log file before the start of the simulation, listed with the associated input information:

- the name of the file where the variable name is referenced; and
- the column number containing the values for the variable, starting with column 1 for the first column after the time stamp.

12.5.2 Description of the Reports Diagnostic Mode

Similarly, when the reports diagnostic mode is selected, the variables defined with the keyword `REPORT` in the problem description and/or the map file (See Section 14.4) are listed with the associated reporting information:

- the name of the file where the variable name is written to; and
- the column number containing the values of the reported variable, starting with column 1 for the first column after the time stamp.

Then, at each report time, the values for the `REPORT` variables are written out to the run log file at the end of the step.

12.5.3 Description of the Convergence Diagnostic Mode

When the convergence diagnostic mode to show the convergence process is selected, *SPARK* writes to the run log file information about the convergence process for the solution of each strongly connected component. The next screenshot shows the convergence diagnostic typically displayed for the solution of a strong component at one time step.

```
STATIC STEP: Problem(room_fc), StepCount(3), Time(360), TimeStep(180)

 --- Component(0) : tol(1e-006), maxtol(0.001), iteration(1...50) ---
  #      Residuals   Increments    Relaxation #Break #Normal Error        Test    Worst-offender variable
P 0      6.0074e-002  N/A          N/A        2      0       6.0000e-002  8e-009  [break] Ta           = 2.500000e+001
  1      2.0262e-010  3.0014e-001  1.00e+000  2      4       2.0598e-001  8e-007  Q_floor              = -2.582216e+002
  2      0.0000e+000  9.5479e-010  1.00e+000  0      0       6.5125e-010  8e-007  T_floor_dot          = -2.582216e-004
```

Step Stamp

The first line is called the step stamp of the diagnostic report:

```
STATIC STEP: Problem(room_fc), StepCount(3), Time(360), TimeStep(180)
```

It indicates:

- the type of step currently being solved, i.e., either a static step or a dynamic step;
- following the tag `Problem`, the name of the problem being solved;
- following the tag `StepCount`, the step count starting at one for the first computed step (usually the initial time solution);
- following the tag `Time`, the current value of the `GLOBAL_TIME` link; and
- following the tag `TimeStep`, the current value of the `GLOBAL_TIME_STEP` link.

Diagnostic reported at a new step always starts with such a step stamp.

Component Stamp

Following the step stamp, diagnostic about the convergence process is reported for each strongly connected component in the order in which they are solved. It starts with the component stamp

```
--- Nonlinear solver for Component(0) : tol(1e-006), maxtol(0.001), iteration(1...50) ---
```

which indicates:

- the evaluation order of the component in the solution sequence generated by *setupcpp*, starting at zero for the first component;
- following the tag `tol`, the value of the prescribed relative tolerance – specified with the key `Tolerance` in the problem preference file –;
- following the tag `maxtol`, the value of the relaxed relative tolerance – specified with the key `MaxTolerance` in the problem preference file –; and
- following the tag `iteration`, the minimum number of iterations to be performed and the maximum number of iterations allowed in the nonlinear solver – specified respectively with the key `MinIterations` and `MaxIterations` in the problem preference file.

Prediction Diagnostic

Then, the convergence diagnostic is reported on a different line for each iteration of the nonlinear solver shown in the column '#'.

The diagnostic begins by reporting the prediction state of the component, identified with the iteration count 0 preceded by the letter 'P'.

```
    P 0      6.0074e-002  N/A          N/A        2      0       6.0000e-002  8e-009  [break] Ta          = 2.500000e+001
```

The prediction state is computed by firing the system of directed inverses comprising the current component, using the predicted values for the break variables. For the prediction iteration, the following diagnostic data is reported:

- shown in the column '`Residuals`', the scaled Euclidean norm of the residual function;
- shown in the column '`#Break`', the number of break variables that failed the prediction convergence test;
- shown in the column '`#Normal`', the number of non-break variables that failed the prediction convergence test; and
- the worst-offender variable during the prediction convergence test with:
    - in the column '`Error`', its convergence error;
    - in the column '`Test`', the tolerance test to be satisfied by the convergence error; and finally
    - in the column '`Worst-offender variable`', its name and current value.

If the worst-offender variable is a break variable, then the variable name is preceded by the tag '`[break]`'.

The tag '`N/A`' in some of the columns indicates diagnostic data that does not apply because it does not make sense in the present context or cannot be calculated for this iteration.

Iteration Convergence Diagnostic

After the prediction diagnostic, the convergence process is reported for each successive iteration of the nonlinear solver.

```
  1      2.0262e-010  3.0014e-001  1.00e+000  2      4       2.0598e-001  8e-007  Q_floor             = -2.582216e+002
  2      0.0000e+000  9.5479e-010  1.00e+000  0      0       6.5125e-010  8e-007  T_floor_dot         = -2.582216e-004
```

The following diagnostic data is displayed:

- the iteration count in the column '#';
- in the column '`Residuals`', the scaled Euclidean norm of the residual function for the current iteration;
- in the column '`Increments`', the scaled Euclidean norm of the iteration increments for both the non-break variables and the break variables;
- in the column '`#Break`', the number of break variables that failed the iteration convergence test;
- in the column '`#Normal`', the number of non-break variables that failed the iteration convergence test; and
- the worst-offender variable during the iteration convergence test with:
    - in the column '`Error`', its convergence error;
    - in the column '`Test`', the tolerance test to be satisfied by the convergence error; and finally
    - in the column '`Worst-offender variable`', its name and current value.

Relaxed Convergence Diagnostic

If convergence against the relative tolerance is not achieved after the maximum number of iterations allowed, *SPARK* performs a convergence test using the relaxed relative tolerance to decide whether or not to proceed with the simulation. The diagnostic report for the relaxed convergence test begins with the letter 'R' and shows the iteration diagnostic data again but for the relaxed relative tolerance.

12.5.4 Description of the Statistics Diagnostic Mode

The statistics diagnostic mode provides information at the end of the simulation on:

- the problem topology (i.e., decomposition of the solution sequence in weak and strong components, number of unknowns and break variables, etc.);
- the numerical performance of the nonlinear solver called by each strong component;
- the numerical performance of the linear solver used by each nonlinear solver; and
- the preference settings used for the solution of each strongly-connected component.

Information such as the number of function evaluations and the average solution times can be used to compare the computational efficiency of the solver. This can be useful to assess the best numerical formulation for a physical model (e.g., generated with different MATCH_LEVEL and BREAK_LEVEL specifications) or the best preference settings to solve a problem.

The number of function evaluations required for the solution of a problem is independent of the hardware configuration and therefore offers a good basis for comparison across multiple platforms, unlike the statistics about the average solution times. However, it should be noted that the computational load associated with a function evaluation depends on the implementation of the inverse function in question. Thus, the statistics about the number of function evaluations do not always reflect the overall solution time. This explains why the statistics log file reports both data to produce an accurate picture of the numerical performance.

13 THE NATIVE INPUT FILE MECHANISM

Values for problem variables can be provided in *SPARK* input files for discrete time stamps specified in strictly increasing order. This mechanism is referred to as the native input file mechanism because it provides supports for reading from *SPARK* input files (See Section 13.3).

To read values from files in a different format, you should use the Read URL mechanism (See Section 14). By default, input values required at runtime will be obtained with the native input file mechanism. However, if a valid Read URL string is specified for a variable, the URL mechanism supercedes the native mechanism for this variable.

It is sometimes more convenient to use multiple input files, thus allowing different time stamp sequences for different set of variables. See Section 7.6 for examples of when this might be useful. The input files are specified in the `InputFiles` segment of the probName.run file. See Section 18 for more details on the format of the run-control file.

At runtime, the *SPARK* input manager opens each of the listed input files and identifies where to search for values for each variable. Then, at every time step the input values are read from the input files and assigned to the variables at the discrete time stamps.

13.1 PRECEDENCE RULE

The input manager does not distinguish between constant and time-varying values. All variables will be sought from the input files specified for the problem.

If a variable does not appear in any input files, then its default value as specified in the problem description will be used instead.

If a variable appears in more than one input file, then the value for the variable will be read from the last occurrence in the list of input files. Therefore, the order in which you specify the input files in the run-control file is important.

13.2 EVALUATION RULE

The input manager always interpolates linearly between the values corresponding to the time stamps specified in the input file around the desired time.

If the desired time is past the last time stamp specified in the input file, then the input manager returns the last specified value without extrapolating.

If the desired time is prior to the first time stamp specified in the input file, then the input manager returns the first specified value without extrapolating.

13.3 FILE FORMAT

To accommodate time-varying inputs, an input file has the tabular form :

```
N          var1       var2       var3       …          varN
t0         val1_0     val2_0     val3_0     …          valN_0
t1         val1_1     val2_1     val3_1     …          valN_1
t2         val1_2     val2_2     val3_2     …          valN_2
*
```

Here var*i* are the variable names and val*i*_*j* are their values at times t*j*, where *i* indicates the column number where to read the value (starting at 0 for the first column containing the time stamps) and *j* indicates the position of the time stamp in strictly increasing order. The entry N in the first column of the first row indicates the number of variables for which values are specified in each following row of the input file.

The final line with only '*' in it is optional and indicates that all values remain fixed from that point forward. This means that the last values defined in the file will be read at each time step past the last time stamp. However, if there is no final line with '*' in the input file, then the input manager will not read the values in the file after the last time stamp.

Constant values have the same value repeated at each time stamp.

13.4 PROPERTY READER

The input manager also allows reading in the properties MIN, MAX, INIT and ATOL from input files for each problem variable, at specified time stamps.

It is recommended to write the values for the different property types in multiple files, where each file contains values only for one property type. Writing the values for the properties in an input file prevents you from having to rebuild the problem when changing the values of an INIT property or of an ATOL property for some variables between successive simulation runs of the same problem.

13.4.1 How to Specify a Property in an Input File

In an input file, the syntax required to indicate a property consists in the name of the variable followed by ':' and the name of the property in question. For example, X:ATOL refers to the property ATOL of the problem variable named X.

Following is an example of an input file that specifies the absolute tolerance values of three variables for the time stamp 0.

```
3          X:ATOL     Y:atol     Z:Atol
0          1.0E-6     1.0E-12    1.0E-4
```

The name of the variable is case sensitive whereas the name of the property is not case sensitive. The property qualified variable names that cannot be parsed for a valid property name by the input manager are reported to the file error.log as warnings. This file should be consulted by the user to identify possible typing mistakes.

13.4.2 When Properties Are Read from Input Files

The input manager reads in the values for the INIT properties for all problem variables only once at the initial time.

The input manager attempts to read in the values for the other properties (i.e., the properties MIN, MAX and ATOL) at the beginning of each time step until the final simulation time is reached.

For example, if the simulation cannot converge because the absolute tolerances specified for some unknown variables appear to be too strict, then it is possible to relax these ATOL values for the time interval in question in order to allow the simulation to proceed past this numerically sensitive phase. The following input file illustrates how to relax the absolute tolerance for the variable X from 10^{-6} to 10^{-4} between the time stamps 10 and 20.

```
1           X:ATOL
0.0         1.0E-6
9.9999      1.0E-6
10.0        1.0E-4
20.0        1.0E-4
20.0001     1.0E-6
```

In order to produce step-like profiles we specify two successive entries for very near time stamps for each change in ATOL values. Note that the difference between the time stamps around the occurrence of the step profile should be smaller than the time increment indicated in the run-control file.

14 THE READ URL MECHANISM

14.1 OVERVIEW AND TERMINOLOGY

The native *SPARK* input file mechanism presented in Section 13 is limited because it only supports reading files with a predetermined format, namely the *SPARK* file format, which places the burden on the user to specify the values using this file format. Sometimes, you want to be able to read values from a file defined using a different format. This situation occurs if the file is readily available from another application and you don't want to or cannot translate it. For example, some application fields might define standard file formats that you need to use in your *SPARK* simulation runs, e.g. such as the weather files in the field of building simulation.

To address these limitations, the native *SPARK* input file mechanism has been extended with the Read URL mechanism. The Universal Resource Locator mechanism is a generalized way of specifying where and how input values are to be obtained at runtime. It is string-based and easily extensible to support more URL handlers that implement new data exchange mechanisms.

Both input mechanisms can be used in the same simulation run, but each variable will seek its input values from either one. If a valid Read URL is specified for a variable then it has priority over the native input file mechanism. Conversely, if no Read URL is specified for a variable, input values will be seeked from *SPARK* input files by default.

Static Read URLs, i.e. URLs specified as part of the problem definition, are specified in the `LINK` statement following the `INPUT` keyword. They can only define for input variables. E.g.,

```
LINK X … INPUT="a valid Read URL string";
```

It is also possible to specify URLs at runtime using a URL map file (See Section 14.4).

There are two main read URL types, *file* and *string*. The subcategories for *file* are DOE-2, TMY and EnergyPlus weather files, columnar file, named column file and formatted file. For the *string* type the subcategories are *schedule* and *algebraic expression*.

All types may be followed with the keyword `interpolate` preceded by the separator '`:`' to force the solver to linearly interpolate between the previous and current values.

In all types, options and specifiers are separated by colons, and with the exception of file names, are case insensitive.

It is the user's responsibility to verify that the units of the data are consistent with their model.

The URL mechanism will be extended in future versions to support Write URLs that will allow reporting values in a different format than with the native *SPARK* output file mechanism (See Section 15.1).

14.2 READ URL FILE TYPE

Following the literal "file" in the first field are the subspecifiers for the file URL type. In the following table, italicized values would be replaced by the desired value, e.g. the actual file name would replace *filename*, and text not in italics is literal, e.g. `doe2bin`.

Table 14-1: Subspecifiers and associated fields for the Read URL file type.

Field 2 (type)	Field 3	Field 4	Field 5	Field 6	Field 7
`doe2bin`	*filename*	*varname1*			
`tmyascii`	*filename*	*varname1*			
`eplusweather`	*filename*	*varname1*			
`column`	*filename*	*# headers2*	*column sep.3*	*column #5*	
`namedcolumn`	*filename*	*# headers2*	*column sep.3*	*# nameline6*	*varname1*
`format`	*filename*	*# headers2*	*format string4*		

Notes:

1. Name of variable desired

2. Number of header lines in file

3. Column separator character

4. String describing output format

5. Desired column number from file

6. Header line containing names of variables

14.2.1 DOE-2 Weather file (doe2bin)

This type will read the DOE-2 binary weather file format. Following the literal `doe2bin` are the file name and the variable name desired, e.g. `dbt` for the dry bulb temperature. The following example would read the air density, `rho` from the DOE-2 weather file for Chicago and interpolate the values from one point to the next:

```
INPUT="doe2bin:chicagotry.bin:rho:interpolate"
```

Table 14-2: Variable names for DOE-2 weather file.

Name	Description	Units
`wbt`	wet bulb temperature	C
`dbt`	dry bulb temperature	C
`barom`	barometric pressure	Pa
`wdir`	wind direction	degrees
`hum`	humidity ratio	-
`rho`	density of air	kg/m^3
`enth`	specific enthalpy	J/kg
`horzrad`	total horizontal radiation	W/m^2
`dirnrad`	direct normal radiation	W/m^2
`wspd`	wind speed	m/s

14.2.2 TMY Weather file (tmyascii)

This type will read the TMY (Typical Meteorological Year) ASCII weather file format. Following the literal `tmyascii` are the file name and the variable name desired, e.g. `diffrad` for the diffuse radiation. The following example would read the dry bulb temperature, dbt from the ASCII TMY weather file for Boston:

```
INPUT="tmyascii:boston.tmy:dbt"
```

Table 14-3: Variable names for TMY weather file.

Name	Description	Units in file	Units returned
extrrad	extraterrestrial radiation	kJ/m^2	J/m^2
dirnrad	direct normal radiation	kJ/m^2	J/m^2
diffrad	diffuse radiation	kJ/m^2	J/m^2
netrad	net radiation	kJ/m^2	J/m^2
globradtlt	global radiation on tilted surface	kJ/m^2	J/m^2
globradhor	global radiation on horizontal surface	kJ/m^2	J/m^2
sunshine	seconds of sunshine	min	s
barom	barometric pressure at sea level	kPa	Pa
stabarom	barometric pressure at station	kPa	Pa
dbt	dry bulb temperature	degrees C x 10	C
dewpt	dew point temperature	degrees C x 10	C
wdir	wind direction	degrees	degrees
wspd	wind speed	m/s x 10	m/s

14.2.3 EnergyPlus Weather File (eplusweather)

This type will read the EnergyPlus weather file format. Following the literal `eplusweather` are the file name and the variable name desired, e.g. `zenithlum` for the zenith illumination. The following example would read the dew point temperature, `dewpt` from the Energy Plus weather file for Los Angeles:

```
INPUT="eplusweather:CA_Los_Angeles_TMY2.epw:dewpt"
```

Table 14-4: Variable names for EnergyPlus weather file.

Name	Description	Units in file	Units returned
year	year	-	-
month	month	-	-
day	day	-	-
hour	hour	-	-
minute	minute	-	-
dbt	dry bulb temperature	C	C
dewpt	dew point temperature	C	C
rh	relative humidity	%	%
barom	barometric pressure at station	Pa	Pa

Name	Description	Units in file	Units returned
exthorrad	extraterrestrial horizontal radiation	Wh/m^2	J/m^2
extdirrad	extraterrestrial direct normal radiation	Wh/m^2	J/m^2
horinfrad	horizontal infrared radiation from sky	Wh/m^2	J/m^2
glohorrad	global horizontal radiation	Wh/m^2	J/m^2
dirnorrad	direct normal radiation	Wh/m^2	J/m^2
difhorrad	diffuse horizontal radiation	Wh/m^2	J/m^2
glohorill	global horizontal illuminance	lux	lux
dirnorill	direct normal illuminance	lux	lux
difhorill	diffuse horizontal illuminance	lux	lux
zenithlum	zenith luminance	Cd/m^2	Cd/m^2
wdir	wind direction	degrees	degrees
wspd	wind speed	m/s	m/s
totalsky	total sky cover	-	-
opaquesky	opaque sky cover	-	-
visibility	visibility	km	m
celheight	ceiling height	m	m
precwater	precipitable water	mm	m
opticdepth	aerosol optical depth	thousandths	thousandths
snowdepth	snow depth	cm	m
lastsnow	days since last snowfall	days	s

14.2.4 Column File

Following the literal `column` are the file name, the number of lines in the header, the column separator and the column number of the data for the variable. For example the following would read column 3 from the file named `mydata.txt` that has 2 header lines with all data separated by commas:

```
   INPUT="file:column:mydata.txt:2:,:3"
```

14.2.5 Named Column File

The difference between the column and named column file types is that the "*#nameline*" (from field 6 in the URL) line of the header names the variables in each column. In the following example, the first `2` is the number of header lines in the file and the second `2` says that the 2nd header line contains the variable names. So this URL would read the 3rd column (because `insolar` is the 3rd variable in the file) from the data file `mydata.txt` with all columns separated by commas:

```
   INPUT="file:namedcolumn:mydata.txt:2:,:2:insolar"
```

with the first few lines of the file containing:

```
The second header line in this file names the variables
time, dbt, insolar, wdir
3:00, 3.4, 33.1, 320
```

14.2.6 Format File

The format file type uses a format string with the same syntax as the `scanf` format string in the *C* computer language to describe the layout of the data file. In the following example the 3rd column is read by skipping over the first two columns (in this case separated with one or more blanks) by using the `%*s` directive and reading the 3rd column with `%lf` (long or double precision floating point):

```
INPUT="file:format:mydata.txt:0:%*s %*s %lf"
```

The percent `%` signals that a formatting character follows. The only format characters that should be used are `s` for string and `lf` for double precision. The asterisk `*` says to skip that field.

14.3 READ URL STRING TYPE

The two read URL string types are schedules and algebraic expression. They are called string types because the data is in the URL string and not read from a file. For this implementation only the DOE-2 type (`doe2sch`) is available for schedule types.

14.3.1 DOE-2 Schedule Type (doe2sch)

The DOE-2 schedule type allows the user to specify different values for a variable in a schedule. For example you may want the lighting level to be a certain value on Monday through Friday from 8am to 5pm and another value the rest of the time and on holidays.

Syntax examples:

```
THRU mon1 day1  (dow1) (h1,h2) (v1,v2,v3) (h3) (v4)
        (dow2) (h4,h5) (v5)
THRU mon2 day2  (dow3) (h6,h7) (v6)
```

Where:

`mon` = Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec

`dow` = ALL HOL WD WE WEH Mon Tue Wed Thu Fri Sat Sun

`HOL` = Holiday

`WD` = WeekDay

`WE` = WeekEnd

`WEH` = WeekEnd + Holiday

`(h1,h2)` means all the hours between `h1` and `h2`. e.g. `(1,3)` means hours 1, 2, 3.

`(v1, v2, v3)` are the values that correspond to the `(h1, h2)` hours list. If the `(v...)` list is shorter than the number of hours given by the `(h1, h2)` list, the last value of the `(v...)` list fills the missing hour slots.

Table 14-5: Holidays in DOE-2 schedule type.

Date	Holiday
January 1	New Year
The third Monday in January	Martin Luther King Jr.
The third Monday in February	President's day

Date	Holiday
The last Monday in May	Memorial Day
July 4	Independence Day
The first Monday in September	Labor Day
The second Monday on October	Columbus Day
November 11	Veteran's Day
The last Thursday in November	Thanksgiving
December 25	Christmas

Example:

```
INPUT="string:schedule:doe2sch:
      thru dec 31 (all) (1,24) (0.1)
      thru jun 30 (wd) (8,17) (0.3)
      thru dec 31 (hol)(1,24)(0.05)"
```

This will give the value of 0.05 for the holidays in the whole year, 0.3 for the hours between 8:00 and 17:00 on weekdays, and 0.1 for the rest of the time. Note the order of the specification is important. The first specification says to use 0.1 for the whole year, but that is overridden for weekdays between 8:00 and 17:00 with 0.3 then any holidays are overridden with 0.05.

The whole URL should be on one line in the .pr, .cc or .cm file. Here it is shown on four lines for visual clarity.

14.3.2 Algebraic Expression Type (expr)

The last string type is the algebraic expression. This provides a fairly versatile way of specifying values that change with time using functions and/or mathematical operators. Here is a table of available mathematical constants, operators and functions. They are all **case sensitive**.

Table 14-6: Functions in Read URL algebraic expression type.

Name	Description
`acos`	arc cosine
`asin`	arc sine
`atan`	arc tangent
`atan2`	arc tangent of y/x
`log`	log base e
`log10`	log base 10
`sin`	sine
`tan`	tangent

Table 14-7: Operators in Read URL algebraic expression type.

Name	Description
`+`	addition
`-`	subtraction
`*`	multiplication

Name	Description
/	division
( )	subexpression grouping
^	power
%	modulus or remainder (e.g. 37 % 7 = 5)

Table 14-8: Constants in Read URL algebraic expression type.

Name	Value	Units	Description
ABS_ZERO	-273.16	C	absolute zero
BOLTZ	$5.67x10^{-8}$	$W/(m^2*K^4)$	Stefan-Boltzmann's constant
CP_AIR	1006.0	J/(kg*K)	specific heat capacity of dry air
CP_VAPOR	1805.0	J/(kg*K)	specific heat capacity of water vapor
CP_WATER	4186.0	J/(kg*K)	specific heat capacity of liquid water
M_G	9.8	m/s^2	gravitational constant
KELV_ZERO	273.16	K	0 degrees C in Kelvin
M_E	2.7182818284590452354	-	*e*
M_LN10	2.30258509299404568402	-	$\log_e(10)$
M_LN2	0.69314718055994530942	-	$\log_e(2)$
M_PI	3.14159265358979323846	-	Π
MW_AIR	0.0289645	kg/mol	molar weight of dry air
MW_WATER	0.01801528	kg/mol	molar weight of liquid water
MW_RATIO	0.62197	-	ratio of molar weights of liquid water over dry air
P_ATM	101325	Pa	atmospheric pressure at sea level
RHO_AIR	1.2	kg/m^3	density of air
RHO_WATER	998.0	kg/m^3	density of water
VISC_WATER	0.001	kg/(m*s)	viscosity of liquid water

Table 14-9: Special variables in Read URL algebraic expression type.

Name	Value	Units	Description
GLOBAL_TIME	-	seconds	current relative *SPARK* time

Example:

```
   INPUT="string:expr:sin(3*(GLOBAL_TIME* M_PI/180)+5)"
```

The above will multiply the current relative time in seconds by Π, divide by `180`, multiply that by `3`, add `5` and take the sine of that.

14.4 URL MAP FILE

The map file is used to translate URLs that are defined in the *SPARK* model to new URLs. This process is referred to as URL string substitution, whereby the URL string specified in the *SPARK* model is substituted with the mapped URL string.

The map file is also used to specify URL strings for the prolem variables at runtime. This is achieved by assigning a new URL string to the name of the variable in question. Conversely, it can be used to clear a URL specification for a variable in order to revert back to the native mechanism for input/report processing (i.e., using the *SPARK* input/report file format). This is done by specifying the empty string `""` after the variable name.

The map file has the name probName.map where probName is the name of the problem being solved. It is read and processed at run time before the simulation phase starts. If no map file exists in the current working directory, then *SPARK* does not perform URL mapping.

14.4.1 The Map File Syntax

The map file contains lines of the form:

```
// Comments
model_url = new_url
variable_identifier = new_url
```

Here `model_url`, `variable_identifier` and `new_url` are quoted strings, or non-quoted strings without spaces in them. If the line starts with a `//`, it is ignored.

URL String Substitution

The field `model_url` contains a valid URL that is specified in the model. It will be substituted with the URL string on the right-hand side, namely `new_url`.

URL String Specification

The field `variable_identifier` contains a variable name prefixed with `r:` or `w:` to indicate read or write context. The field `new_url` contains a valid URL specification or an empty string `""` or the tag "REPORT". The URL specified here becomes the new URL for the variable with the corresponding read/write context. When the `new_url` contains an empty string, it removes the URL that was specified in the model, and reverts the read/write processing to native input/report file format.

The following code snippet shows the possible content of a map file. The first entry specifies a string substitution rule for the model URL `"string:expr:sin(3*(GLOBAL_TIME*M_PI/180)+5)"` that will be replaced with the new URL string `"string:expr:cos(2*(GLOBAL_TIME/180))"`.

Next are two URL string specifications, one for the write context, one for the read context. The variable `mass_flow` will be reported to the output file and the variable `obj1~Tin` will get its value from the field `dbt` in the EnergyPlus weather file named USA_NV_Las.Vegas_TMY2.epw.

```
// comment line
"string:expr:sin(3*(GLOBAL_TIME*M_PI/180)+5)" = "string:expr:cos(2*(GLOBAL_TIME/180))"
"w:mass_flow" = "REPORT"
"r:obj1~Tin" = "file:energyplusweather:USA_NV_Las.Vegas_TMY2.epw:dbt:interpolate"
// end
```

14.4.2 Loading Rules

In the map file processing, the following rules are applied:

1. If the variable has a model URL (i.e., a URL string is specified for this variable in the *.pr or *.cm files):
    - The map file is searched for `model_url`.
    - If a match is found the replacement `new_url` is substituted.
    - If `new_url` is an empty string, the rule 2 is also tried.
    - Otherwise, `new_url` becomes effective.
2. The variable name together with its read or write context is searched in the map file entries for `variable_identifier` with `r:` or `w:` prefixes. If a match is found, the replacement `new_url` is substituted.

The current version of *SPARK* does not implement any Write URL handlers yet. Therefore, the only reporting mechanism that is available is the native reporting mechanism described in Section 15.1. However, the map file can be used to tag the problem variables that need to be reported by specifying the string `"REPORT"` for the `new_url` string in the write context.

15 OUTPUT AND POST PROCESSING

15.1 THE OUTPUT FILE

When *SPARK* runs there is output to the screen and to an output file with extension .out. The screen output is primarily for visual feedback, letting you know where *SPARK* is in processing your problem. The output file contains results of the numerical solution process at each time step. The format of the output file is exactly like that of input files, i.e.,

```
n       label   label   label
t0      value   value   value
t1      value   value   value
etc.
```

where n is the number of reported variables, each label is a problem variable with the REPORT keyword expressed in the problem file, and each value is the value for the corresponding variable at the time stamp t*i*.

It is possible to use the URL map file (See Section 14.4) to tag problem variables with the REPORT keyword so that they will be reported to the output file. This mechanism lets you specify which variables need to be reported without having to re-build the problem.

15.2 PLOTTING THE OUTPUT FILE

The output of *SPARK* can be read by conventional spreadsheet and plotting programs. If you use Microsoft *Excel* or a similar program, simply open the *SPARK* output file into a worksheet and use space as the delimiting character between fields. This will place your output neatly into rows and columns, from which you can construct plots (charts) in the usual *Excel* manner.

VisualSPARK provides options that auto-make the graphing process (see *VisualSPARK Users Guide*). If you use *gnuplot*, a program called *makegnu* is provided with *WinSPARK* that will generate an input file for that program.[42] To use *makegnu*, type:

```
makegnu room_fc.out room_fc.plt <enter>
```

The output file, room_fc.gnu, will contain the *gnuplot* commands, e.g.:

```
set data style linesset xlabel "time"
set ylabel "mcp"
plot "room_fc.out"  using 1:2 notitle
pause -1 "Press <enter>"
set ylabel "Q_flow"
plot "room_fc.out"  using 1:3 notitle
pause -1 "Press <enter>"
set ylabel "Ta"
plot "room_fc.out"  using 1:4 notitle
pause -1 "Press <enter>"
set ylabel "T_floor"
plot "room_fc.out"  using 1:5 notitle
pause -1 "Press <enter>"
```

[42] Although not provided in the VisualSPARK release, *makegnu* is available free from Ayres Sowell Associates, Inc. and will run on UNIX as well as Windows platforms.

Then to plot with *gnuplot*, type

```
gnuplot room_fc.plt <enter>
```

This assumes you have *gnuplot* in your command path. More elaborate plots, combining several results on the same plot, for example, can be done by editing the *gnuplot* input file, or by running *gnuplot* interactively. The *gnuplot* documentation should be consulted for more information.

15.3 POST PROCESSING IN *MATLAB*

The *SPARK* distribution comes with *MATLAB* script files that help loading the *SPARK* data files into arrays in the *MATLAB* environment. These files are located in the utils/matlab subdirectory where *SPARK* is installed.

Table 15-1: *MATLAB* script files for post-processing of *SPARK* files.

Script Name	Description
LoadSPARKFile.m	Loads a file in *SPARK* format (i.e., input, output, trace, or snapshot file) into *MATLAB* arrays with the time stamps (first column), the values at each time stamp in each column and the names of the variables in each column.
DiffSPARKFiles.m	Compares the numerical values contained in two files with the same format against the prescribed precision and detects the offending entries.
FindName.m	Returns the position in the variable names array of the entry matching the target variable name. If not found returns 0. Convenient to identify in which column of a *SPARK* file a particular variable can be found.
LoadOneJacobian.m	Loads the Jacobian matrix contained in a Jacobian trace file at a specific iteration into a *MATLAB* array.
ComputeJacobianCondition.m	Loads the Jacobian matrix contained in a Jacobian trace file at each iteration and computes its condition number.

16 LOG FILES

SPARK generates various log files over the course of the simulation that contain specific information about the solver operation. These files should be consulted by users to gain deeper knowledge about the internal operations and the numerical behavior.

16.1 RUN LOG FILE

If the diagnostic level (See Section 12.5) specified in the run-control file is not silent, then *SPARK* generates a file named run.log in the current working directory. This file contains the desired diagnostic information about the simulation run.

16.2 ERROR LOG FILE

If any errors and/or warnings occur during the simulation, *SPARK* generates a file named error.log in the current working directory. This file contains detailed explanations for the error/warning, starting with the corresponding time stamp. The cause of an error/warning can be either numerical (e.g., detection of no convergence or of a singular linear system) or internal (e.g., cannot open an input file or create an output file, cannot parse a variable name in an input file, cannot allocate memory on the heap, etc.).

A warning indicates a situation that might result in an error and that therefore needs to be brought to the attention of the user. A warning message starts with the tag '`[WARNING]`'.

An error results in the abnormal termination of the solver. An error message starts with the tag '`[ERROR]`'.

If no error occurred, then no error log file is generated. If an error log file has been generated and the simulation run has been successful, then you should consult it and make sure that the reported warning messages do not have any impact on the solution. If the simulation run has not been successful, then the messages in the error log file should help you identify the cause of the error.

16.3 FACTORY LOG FILE

When the *SPARK* problem is loaded at runtime from a file, the runtime loader generates a file named *probName*.factory.log in the current working directory, where *probName* stands for the name of the problem being loaded. This file contains information about the problem description and the loading times for each section of the description.

If the runtime problem loader fails, this file should be consulted to identify the possible cause.

16.4 DEBUG LOG FILE

The *SPARK* solver can be used in debug mode by linking the problem driver to the solver library compiled with the preprocessor macro `SPARK_DEBUG` being defined.

The *SPARK* makefile produces a build with debugger information when:

- the environment variable `DEBUG=yes` is defined; or
- the *gmake* program is run with `DEBUG=yes` at the command line,

  ```
  gmake <target> DEBUG=yes   <enter>
  ```

  where `<target>` stands for any valid target defined in the *SPARK* makefile.

When running in the debug mode, *SPARK* generates a file called debug.log in the current working directory. This file contains detailed information about every phase of the simulation process:

- loading the problem description,
- initializing the problem with the specified runtime controls,
- loading the past values for all problem variables from the input files,
- solving the problem at the initial time, and
- solving the problem until the final time.

In particular, you can trace the operation of the input manager to identify:

- from which input file and column the problem variables get their values, and
- which new values and properties are read at each time step for which variables.

The debug log file should be consulted in case of abnormal termination of the solver operation as it may help you find the cause of the error. Therefore, if a simulation fails, it is recommended to run the simulation again in the debug mode to generate the debug log file.

By default, *SPARK* does not run in debug mode in order to avoid the performance penalty incurred from the extensive output to the log file.

16.5 BACKTRACKING LOG FILE

When a strongly-connected component in the *SPARK* problem is solved using a backtracking method, the nonlinear solver generates a file named *probName.id.*backtracking.log in the current working directory, where *probName* stands for the name of the problem being loaded and *id* for the evaluation number of the component in question (starting at 0 for the first component in the solution sequence). This file contains self-describing information in tabular form about the backtracking process at each step for each iteration.

If the component in question fails to converge, this file should be consulted to analyze the convergence process and possibly identify the cause of the non-convergence.

Note that the backtracking log file is only generated in the debug mode.

17 SNAPSHOT FILES

17.1 WHY SNAPSHOT FILES ARE USEFUL

There are occasions on which you may want to stop a simulation, then restart it from the same point at a later time. This need can arise when the problem experiences a long run time, or a difficult solution. Or, you may want to repeat a simulation using precisely the same initializations of dynamic and break variables. These techniques are supported in *SPARK* with the notion of snapshot files. You can request that snapshot files be generated at `InitialTime` and/or `FinalTime` (see Section 18) as discussed below. It is also possible to generate snapshot file at any other point of the simulation by sending a snapshot request from a callback function (See Section 10.2).

A snapshot file contains the values of all problem variables for the last four time stamps in a format identical to that of a normal output report. And, because *SPARK* input files and output files have the same format, you can specify a snapshot file as an input file in a subsequent run of the same problem to restart the problem.

Since a snapshot file contains the values for all the problem variables (not just those that were tagged with the `REPORT` keyword in the problem definition file), it is a very powerful reporting and diagnostic mechanism as well as serving as initialization files for restarting.

17.2 GENERATING SNAPSHOT FILES

You request generation of snapshot files by specifying corresponding keys in the run-control file (see Section 18), along with the desired name for the snapshot file. Two keys are available, `InitialSnapshotFile` and `FinalSnapshotFile`. The values of these keys should be the paths to the files where you want the results saved. For example, if you want both initial and final snapshot files, your run-control file probName.run must contain the following two entries:

```
InitialSnapshotFile ( probName.init () )
FinalSnapshotFile   ( probName.snap () )
```

The key `InitialSnapshotFile` generates a snapshot file with the initial time solution in probName.init, whereas the key `FinalSnapshotFile` generates a snapshot file with the solution at the final time in probName.snap.

Note that the file names, including the extensions, are arbitrary, i.e., you can use whatever extension you wish.

Normally, you will want to include the file path to specify where it is to be saved. In the example, it is saved in the current working directory.

17.3 USING SNAPSHOT FILES TO INITIALIZE A SIMULATION RUN

17.3.1 Specifying Snapshot Files as Input Files

To use a snapshot file for initializing a subsequent run you simply specify it in the `InputFiles` segment in the run-control file, with the other input files.

For example, to restart your problem initialized from the final solution of the previous run, captured in the file probName.snap, in the file probName.run modify the `InputFiles` segment to read :

```
  InputFiles (
     probName.snap ()
     probName.inp ()
```

```
)
```

Also, you will need to set the `InitialTime` key in the run-control file to the value at which you want to restart the simulation, usually the last time stamp appearing in the snapshot file.

The order in which the input files are specified in the `InputFiles` segment is important. Specifying the snapshot file first ensures that the values specified in the file probName.inp correctly overwrite the values (specified for the same variables) that appear in the previously declared snapshot file probName.snap.

To satisfy the precedence rule of the input manager (see Section 13.1) and to follow the categorization of the different types of inputs proposed in Section 7.6.1, the order in which the different types of input files are be specified under the `InputFiles` segment in the run-control file is:

1. files with the predicted initial values for the break variables;
2. files with the initial values for the dynamic variables;
3. snapshot files with values to restart the simulation;
4. files with the values for the input variables (constant and time-varying).

17.3.2 Restarting after a Numerical Error

If the key `FinalSnapshotFile` was specified, in the event of a non-convergence or other solution failure such as bad numerics, the snapshot file will be generated automatically at the time when the failure occurred (instead of the final time) along with the values for the last four time stamps.

This provides values of all variables at the point of non-convergence, which might be helpful in discovering the reasons for the non-convergence. Also, the snapshot file can be used to restart the simulation with values at the last valid time stamp and with modified problem preference settings and/or new predicted values for the break variables in an attempt to fix the numerical problem.

17.3.3 Enforcing Initial Conditions from a Different Problem Definition

Another way to use a snapshot file to initialize a problem is to first solve a static problem (no integrators) derived from the dynamic problem and with initial conditions for some of the unknowns of the dynamic problem (specified as inputs to the static problem). The resulting snapshot file of the solution of the static problem can then be used to start the dynamic problem with the desired initial conditions enforced.

18 RUN-CONTROL FILE

We introduced the *SPARK* run-control file, probName.run, in the examples in Section 2. There, we were concerned with only the basic, required elements of this file needed to run simple problems. In this Section we will examine the run-control file further, showing the format as well as all aspects of a *SPARK* run that can be controlled from it.

The run-control information needed for a *SPARK* problem comprises the keys and values as shown in Table 18-1. Items shown in boldface are required.

Table 18-1: Run Controls

[Key] in run-control file	**Definition**	**Typical value**
[**InitialTime**]	The time at which the simulation begins.	0.0
[**FinalTime**]	The time at which the simulation ends.	0.0
[**InitialTimeStep**	The initial time span between solution points.	1.0
[VariableTimeStep]]	If set to 1, then the time step will be adapted during the course of the simulation if necessary. Otherwise, the time step remains constant.	0
[MinTimeStep]	Minimum allowed time step. Only used if VariableTimeStep is set to 1.	1.0E-6
[MaxTimeStep]	Maximum allowed time step. Only used if VariableTimeStep is set to 1.	+1.0E-6
[ConsistentInitialCalculation]	If set to 1, then *SPARK* solves a static step at the initial time to ensure consistent initial values.	1
[FirstReport]	The time at which the first output is desired.	0.0
[ReportCycle]	The time interval between output reports.	0 (= report all solution points)
[DiagnosticLevel]	Level of diagnostic output desired (see Section 12.5).	0 = silent
[InputFiles]	List of input files with paths (see Section 7.6).	probName.inp c:\Phoenix\weather.inp
[OutputFile]	Output file with path (see Section 14).	probName.out
[InitialSnapshotFile]	Initial time snapshot file with path (see Section 17).	probName.init
[FinalSnapshotFile]	Final time snapshot file with path (see Section 17).	probName.snap

In the current *SPARK* release, the time step remains constant by default during the course of the simulation. Therefore, the value of the key `InitialTimeStep` reflects the constant time step. If the key `VariableTimeStep` is set to 1, then the time step will be adapted whenever it is necessary during the course of the simulation. In this case, the key `InitialTimeStep` specifies the time step to use initially.

Also, when the key `ConsistentInitialCalculation` is set to 1, then the *SPARK* solver starts the simulation with a static step as explained in Section 10.3. Setting this key is equivalent to sending a restart request before the first simulation step.

The run-control information is stored in the file probName.run using the preference file format, described in Appendix C. A typical run-control file is then:

```
(
   InitialTime           ( 0.0 ())
   FinalTime             ( 5.0 ())
   InitialTimeStep       ( 0.1 ())
   FirstReport           ( 0.0 ())
   ReportCycle           ( 0.1 ())
   DiagnosticLevel       ( 3 ())
   InputFiles            (   frst_ord.inp ()
                             frst_ord_ic.inp ()
                         )
   OutputFile            (   frst_ord.out ())
   InitialSnapshotFile   (   frst_ord_dyn.init ())
   FinalSnapshotFile     (   frst_ord_dyn.snap ())
)
```

19 *SPARK* LANGUAGE REFERENCE

19.1 NOTATION USED IN THIS SECTION

1. KEYWORDS are shown uppercase, although they are case insensitive in the language.
2. ♦ means required entry.
3. name_or_par means a name or parameter name. The parameter name must have a substitution-value that is a valid name.
4. val_or_par means value or parameter name. The parameter name must have a substitution-value that is a valid numeric value.
5. Items separated by | means choose one of the items; for example, <x|y|z> means x or y or z.
6. An item inside question marks, e.g., ?connections1?, is defined later in the construct in which it appears.
7. <item> means the item is optional.
8. Definition of higher level and lower level: Problem level is the highest, any object declared inside the problem file is the next lower level, etc. When referring to hierarchy, the problem is the highest level and the atomic class is the lowest.

19.2 SPECIAL CHARACTERS

Special characters are those used by the *SPARK* parser to identify parts of the language. They should not be part of user names.

1. Used in *SPARK* syntax: " # ( ) , . ; = [ ] ' ` { } ~ SPACE TAB NL (newline) /* //
2. Delimiters: SPACE TAB NL. More than one of these characters or combination are ignored.
3. The statement terminator is the semicolon (;).

19.3 NAMES AND OTHER STRINGS

19.3.1 Reserved Names

#endif	DECLARE	KEYWORDS	PORT
#ifdef	DEFAULT	LIKE	PREDICT_FROM_LINK
ABSTRACT	EQUATIONS	LINK	PROBE
ABSTRACT_END	FUNCTIONS	MATCH_LEVEL	REPORT
ATOL	GLOBAL_TIME	MAX	UPDATE_FROM_LINK
BAD_INVERSES	GLOBAL_TIME_STEP	MIN	(same as INPUT_FROM_LINK)
BREAK_LEVEL	INIT	NOERR	VAL
CLASSTYPE	INPUT	PARAMETER	
CONNECT_HINT	INPUT_FROM_LINK	PAST_VALUE_ONLY* (not yet implemented)	

Note: Reserved names are case insensitive, except for #ifdef and #endif.

19.3.2 Rules for User-Specified Names

1. They must not contain any reserved characters.
2. They must not begin with a digit (0 – 9).
3. They are case sensitive.
4. They may not be the same as reserved names.
5. They can be of any length.

19.3.3 Literals

User-specified literal strings are enclosed inside double quotes, e.g., `"This is a literal."` They can contain any character except the double quote (").

19.4 COMMENTS

There are two kinds of comments:

1. `/*comment...*/` *C*-like comment
2. `//comment...` *C++* style comment to end of line

19.5 STATEMENT TERMINATOR

Statement terminator is the semicolon (;).

19.6 COMPOUND STATEMENT

A compound statement is delimited by curly braces: `{ ... }`. Examples of compound statements are `FUNCTIONS` (Section 19.17) and `EQUATIONS` (Section 19.16).

19.7 ATOMIC CLASS FILE

The *SPARK* atomic class is the smallest modeling element. Atomic classes may be used directly in problem files or combined into macro classes to form larger modeling elements.

File name convention: class_name.cc

Format:

```
------------------file class_name.cc    begin---------------------
/* CLASS  class_name   "description..."  KEYWORDS=keyword1,...;
   ABSTRACT
*/
#ifdef SPARK_TEXT  ♦
CLASSTYPE  statements
PARAMETER  statements
PORT       statements  ♦
EQUATIONS { equation statements }
FUNCTIONS { function statements }  ♦
#endif /*SPARK_TEXT*/  ♦
#include "spark.h"  ♦
   callback c++ functions go here
------------------file class_name.cc    end----------------------
```

Notes:

1. PARAMETER statements must appear before they are referenced.
2. PORT statements must appear before EQUATIONS and FUNCTIONS statements.
3. While the material in the /*...*/ header is ignored by the parser, it may be used by browsers and/or utility programs.

19.8 MACRO CLASS FILE

A *SPARK* macro class connects atomic and other macro classes to form larger modeling elements.

File name convention: class_name.cm

Format:

```
------------------file class_name.cm    begin--------------------
/* CLASS_MACRO  class_name   "description..."   KEYWORDS=keyword1,...;
ABSTRACT
*/
PARAMETER  statements
PORT       statements ♦
PROBE      statements
DECLARE    statements ♦
LINK       statements ♦
------------------file class_name.cc    end----------------------
```

Notes:

1. PARAMETER statements must appear before they are referenced.
2. PORT statements must appear before any DECLARE or LINK statements.
3. DECLARE statements must appear before any LINK statements that refer to the objects defined by DECLAREs.
4. While the material in the /*...*/ header is ignored by the parser, it may be used by browsers and/or utility programs.

19.9 PROBLEM FILE

The *SPARK* problem file combines macro and/or atomic classes to form the largest modeling element.

File name convention: problem_name.pr.

Format:

```
------------------file problem_name.pr    begin--------------------
/* PROBLEM  class_name   "description..."   KEYWORDS=keyword1,...;
ABSTRACT
*/
PARAMETER  statements
PROBE      statements
DECLARE    statements  ♦
LINK       statements  ♦
------------------file class_name.cc    end----------------------
```

Notes:

1. `PARAMETER` statements must appear before they are referenced.
2. `DECLARE` statements must appear before any `LINK` statements that refer to the objects defined by `DECLARE`s.
3. While the material in the `/*...*/` header is ignored by the parser, it may be used by browsers and/or utility programs.

19.10 PORT STATEMENT

The PORT statement describes an externally visible connection point (interface variable) of a class. When an object is instantiated from a class by a DECLARE statement, connections can only be made to its ports.

The PORT statement has two forms :

1. Atomic port does not have subports.
2. Macro port has subports.

19.10.1 Atomic port

An **atomic port** has the form:

```
PORT port_name   ♦
            [unit]
            "description..."
            ATOL=val_or_par
            BREAK_LEVEL=val_or_par
            CONNECT_HINT="-class1.portx,class2.porty"
            DEFAULT=val_or_par
            INIT=val_or_par
            LIKE=anotherPortName
            MATCH_LEVEL=val_or_par  ;
            MAX=val_or_par
            MIN=val_or_par
            NOERR
```

Here:

port_name	:	Name of the port; must not contain any special characters (see Section 19.2).
[units]	:	Units of the port. Used to give a warning if a variable with different units is linked to this port.
"description..."	:	Short description of the port. This field is used by browsers.
BREAK_LEVEL	:	The default break level values for connections to this port.
CONNECT_HINT	:	Used by browsers to determine acceptable connections. "-class1.portx, class2.porty"means that connecting this port to **portx** of any instance of **class1** is not permitted, but connecting this port to **porty** of any instance of **class2** *is* encouraged. For acceptability of a connection, first units, then CONNECT_HINTs is checked.
DEFAULT	:	If this subport is not connected, it behaves as if its value is fixed at val_or_par.

LIKE=anotherPortName	:	All of the properties (except the description fields) from the previously defined port named `anotherPortName` are copied to the current port. The copied properties include the subports.
		Note that any other input specified in the current port statement overrides the copied information. The example below shows port statements using the `LIKE` keyword:
		`port aa “description of aa” [deg_C] MIN=-5 MAX=20 ;`
		`port bb “description of bb” [deg_C] LIKE=aa MIN=0 ;`
		produce the same specifications as:
		`port aa “description of aa” [deg_C] MIN=-5 MAX=20 ;`
		`port bb “description of bb” [deg_C] MIN=0 MAX=20 ;`

ATOL, INIT, MIN, MAX	:	Absolute tolerance, initial, minimum, and maximum values assigned to variable created by connections to this port. Higher-level settings will take precedence.
MATCH_LEVEL	:	The default match level values for connections to this port.
NOERR	:	Do not give error message if this port is not connected when this class is used (instantiated). Allows ports that can be optionally used.

19.10.2 Macro port

A **macro port** is composed of two or more subports (see Section 8.1); it has the form:

```
PORT port_name    ♦
                [unit1]
                "port description..."
                CONNECT_HINT="-class1.portx,class2.porty"
                LIKE=anotherPortName
                NOERR
                [unitOfPort]
       ,  .subport_name1    ♦
                [unit2]
                "subport description..."
                ATOL=val_or_par
                BREAK_LEVEL=val_or_par
                DEFAULT=val_or_par
                INIT=val_or_par
                MATCH_LEVEL=val_or_par
                MAX=val_or_par
                MIN=val_or_par
       ,  .subport_name2
                   ...etc ...
       , ...            .
       ;
```

Here:

`port_name`	:	Name of the port; must not contain any special characters (see Section 19.2).
`[units]`	:	Units of the port. Used to give a warning if a variable with different units is linked to this port.
`"description..."`	:	Short description of the port. This field is used by browsers.
`CONNECT_HINT`	:	Used by browsers to determine acceptable connections. `"-class1.portx, class2.porty"`means that connecting this port to **portx** of any instance of **class1** is not permitted, but connecting this port to **porty** of any instance of **class2** *is* encouraged. For acceptability of a connection, first units, then `CONNECT_HINT`s is checked.
`NOERR`	:	Do not give error message if this port is not connected when this class is used (instantiated). Allows ports that can be optionally used.

`LIKE = anotherPortName`	:	All of the properties (except the description field) from the port named `anotherPortName` are copied to the current port. The copied properties include the subports. Any other input that is specified in the current port statement overrides the copied information.

For example, the following two macro port statements:

```
PORT AirEnt1 "Inlet air stream 1" [airflow]
            .m "air mass flow" [kg_dryAir/s]
          , .w "hum. ratio" [kg_water/kg_dryAir]
          , .h "enthalpy" NOERR [J/kg_dryAir] ;
PORT AirEnt2 "Inlet air stream 2"  LIKE=AirEnt1 ;
```

are equivalent to:

```
PORT AirEnt1 "Inlet air stream 1" [airflow]
            .m "air mass flow" [kg_dryAir/s]
          , .w "hum. ratio" [kg_water/kg_dryAir]
          , .h "enthalpy" NOERR [J/kg_dryAir] ;
PORT AirEnt2 "Inlet air stream 2"  [airflow] ;
            .m "air mass flow" [kg_dryAir/s]
          , .w "hum. ratio" [kg_water/kg_dryAir]
          , .h "enthalpy" NOERR [J/kg_dryAir] ;
```

`.subport_name`	:	Name of the subport. Note the leading period. If the subport contains other subports, this is specified as `.subport_name.subport_of_subport_name`. The `subport_of_subport_name` is specified for each `subport_of_subport`. For example, if we have port x with subports `a,b` and subport `a` has its subports `a1,a2` then we write:

```
PORT x  ...etc...
       .a.a1  ...etc...
       .a.a2  ...etc...
       .b  ...etc...
```

`.subport_name`	:	Name of the subport. Note the leading period. If subport contains other subports, this specified as `.subport_name.subport_of_subport_name`. Note that `subport_of_subport_name` is specified for each `subport_of_subport`; e.g. If we have port `x` with subports `a,b` and subport `a` has its subports `a1,a2` we write:

```
PORT  x  ...etc...
        .a.a1  ...etc...
      , .a.a2  ...etc
      , .b  ...etc...  ;
```

`BREAK_LEVEL`	:	The default break-level values for connections to this subport.
`DEFAULT`	:	If this subport is not connected, it behaves as if its value is fixed at `val_or_par`.
`ATOL`, `INIT`, `MIN`, `MAX`	:	Default absolute tolerance, initial, minimum, and maximum values assigned to variables created by connections to this port.
`MATCH_LEVEL`	:	The default match level values for connections to this subport. See Section 12.3.

19.11 PARAMETER STATEMENT

The PARAMETER statement is used to assign a numeric or symbolic value to a name. When this name is used in any place that can take the parameter name, the value of the parameter is substituted in place of the name. For example the following two statements:

```
PARAMETER abc = 12.3 ;
PORT      x  INIT=abc ;
```

Are equivalent to:

```
PORT x  INIT=12.3 ;
```

The parameter statement has the form:

```
PARAMETER name1 = substitution_value1, name2 = substitution_value2, ...;
```

If a problem and one of its classes have parameters of the same name, the value of the problem's parameter is used. Similarly, if a macro and one of its classes have parameters of the same name, the value of the macro's parameter is used. That is, higher level PARAMETER definitions take precedence.

19.12 PROBE STATEMENT

Without the PROBE statement, lower level links (e.g., those in a macro object) are not visible at higher levels (e.g., in the problem file) unless they are connected through ports. The PROBE statement is provided to allow higher-level assignment of values to certain keywords of lower-level links. It can also be used to report such links. See Section 8.5 for examples.

The PROBE statement has the form:

```
PROBE  name    <?port_resolution? | ?link_resolution?>  ♦
               ATOL=val_or_par
               BREAK_LEVEL=val_or_par
               INIT=val_or_par
               INPUT
               MATCH_LEVEL=val_or_par
               MAX=val_or_par
               MIN=val_or_par
               PREDICT_FROM_LINK=<?port_resolution? | ?link_resolution?>
               REPORT
               INPUT_FROM_LINK=<?port_resolution? | ?link_resolution?>
               VAL=val_or_par ;
```

Here:

`name`	:	Name of probe.
`?port_resolution?`	:	Concatenated object name followed by `port.subport...` name that uniquely identifies the connection. It has the form: `obj1`obj2...port.subport.subport_of_subport...`
`?link_resolution?`	:	Concatenated object name followed by `~.` followed by link name followed by `subport...` of link that uniquely identifies the link. It has the form: `obj1`obj2...~link5.subport.subport_of_subport...` For problem level links, it has the form: `~link5.subport.subport_of_subport...`
`ATOL`, `INIT`, `MIN`, `MAX`, `BREAK_LEVEL`, `MATCH_LEVEL`, `INPUT`, `REPORT`, `INPUT_FROM_LINK`, `PREDICT_FROM_LINK`, `VAL`	:	Same as for `LINK` statement (see Section 19.14).

19.13 DECLARE STATEMENT

The DECLARE statement is used to instantiate a class, creating one or more objects. It has the form:

```
DECLARE  name_or_par   obj_name1
                     , obj_name2
                     , ... ;
```

Here obj_name can be either a valid name or a PARAMETER name that defines a valid name.

19.14 LINK STATEMENT

The `LINK` statement is used to make connections between ports of objects instantiated in this class and/or ports of this class. It has the form:

```
LINK  name  "link_description"   ?entries1? , ?entries2? , ...
                , (.sublink1...){ ?entries3? , ?entries4? , ... }
                , (.sublinkN...){ ?entriesM? , ... } ;
```

The optional `(.sublink1...){ ... }` form means that the entries inside `{}` apply to the *.sublink1...* component of the macro-link. Here, *.sublink...* is a valid *.portal...* name for this link.

The `?entriesX?` contains items from the following: (if the `?connection?` item is not present, the `LINK` statement must have one of `INPUT, INPUT_FROM_LINK, GLOBAL_TIME, GLOBAL_TIME_STEP` attributes) where at least the `?connection?` item must be present:

```
        < ATOL = val_or_par >
        < INIT = val_or_par >
        < INPUT >
        < INPUT_FROM_LINK = linkFrom | linkFrom.sublink... >
        < GLOBAL_TIME | GLOBAL_TIME_STEP >
        < MAX  = val_or_par >
        < MIN  = val_or_par >
        < PREDICT_FROM_LINK = linkFrom | linkFrom.sublink... >
        < REPORT >
        < VAL  = val_or_par >
      < ?connection? >
          < BREAK_LEVEL = val_or_par >
          < MATCH_LEVEL = val_or_par >
```

Note that: `INPUT`, `PREDICT_FROM_LINK`, `INPUT_FROM_LINK`, `GLOBAL_TIME`, `GLOBAL_TIME_STEP` qualifiers are mutually exclusive; only one of them may be specified in a `LINK` statement.

Here:

`name`	:	Link name.
`"link_description"`	:	Description , used by browsers.
`ATOL = val_or_par`	:	The absolute tolerance value specified for the variable created by this link (see Section 11.7.1).
`INIT = val_or_par`	:	Gives initial value to the variable. If the variable referenced by this link is a break variable the value is used only once, in the first Newton-Raphson iteration.
`INPUT`	:	Input the variable created by this link, using link name as input variable name.
`INPUT_FROM_LINK`	:	Makes the variable that is created by a current link statement a Previous-Value Variable, see Section 8.3. The value of the variable remains the same during Newton-Raphson iterations (i.e., it is treated as if input. At the beginning of the time step, before solving the problem equations, the saved previous value of `linkFrom` is assigned to a variable defined by the current link statement.
`GLOBAL_TIME`	:	Connects the variable that is referenced by this link to the calculation time value

		that is specified by run control data.
GLOBAL_TIME_STEP	:	Connects the variable that is referenced by this link to the calculation time step value that is specified by run control data.
MIN, MAX	:	Give min, max values to the variable created by this link.
PREDICT_FROM_LINK= linkFrom	:	If the variable referenced by this link is a break variable, give initial value to it from the current value of `linkFrom`. Unlike the `INIT` keyword, `PREDICT_FROM_LINK` supplies the initial value that is copied from `linkFrom` for Newton-Raphson for every time step.
REPORT	:	Output the variable referenced by this link, using link name as report variable name.
VAL = val_or_par	:	Set the value of the variable defined by this link to a constant value `val_or_par`. It assigns the constant value, as if it is input, to the variable defined by the `LINK` statement. If, in the same `LINK` statement, there are connections to the ports of this class then this value can propagate to outside of this class. This value can be overridden later by the `INPUT, GLOBAL_TIME...` keywords referencing the same variable at higher levels.

?connection?	:	This specifies either `.port_of_this_class` including the resolution of the subport if necessary e.g.,
		`.port_of_this_class`
		`.port_of_this_class.subport...`
		or connection to a port of an object declared in this class including the resolution of the subport, e.g.
		`object.port`
		`object.port.subport`

BREAK_LEVEL	:	Break level given to this connection.
MATCH_LEVEL	:	Match level given to this connection.

19.15 INPUT STATEMENT

The `INPUT` statement is exactly like a `LINK` statement for which the `INPUT` keyword has been specified.

19.16 EQUATIONS STATEMENT

The `EQUATIONS` statement within an atomic class (see Section 19.7) specifies the equations that are used to generate the *C++* functions of the class. This statement is currently used by browsers and symbolic processors only. It is a compound statement (see Section 19.5). An example is:

```
EQUATIONS {
            p1.a = x ;
            p1.b = y ;
            p2   = z ;
            x = y^2 * z^2 , x > 0 ;
            BAD_INVERSES = y, z ;
          }
```

Notes:

1. In the above example, `x`, `y` and `z` are "helper" symbols that simplify the equation. The notation `p1.a` means the **a** subport of port **p1**. The example shows the equation relating `x`, `y` and `z` (i.e., $x = y^2 \cdot z^2$) and indicates that `x` is restricted to positive values.
2. An atomic class can only have one equation that shows the relationship between the ports of the class.

19.17 FUNCTIONS STATEMENT

The FUNCTIONS statement without an atomic class (see Section 19.7) specifies the *C++* inverse functions associated with the ports of the class. It is a compound statement (see Section 19.5) of the form:

```
FUNCTIONS {
            DEFAULT_RESIDUAL=default_residual_fun( port1,....,portN ) ;
            port1 <,port2,port3,....> = explicit_fun1( port2,.... )
                                      <method1 = method_fun1(port2,... )
                                       method2 = method_fun2( ... )
                                       ....> ;
          }
```

Here `method1,method2...` are keywords from the following list that specify the callback points during a *SPARK* run.

CHECK_INTEGRATION_STEP	PREDICT	STATIC_COMMIT
COMMIT	PREPARE_STEP	STATIC_CONSTRUCT
CONSTRUCT	RESIDUAL	STATIC_DESTRUCT
DESTRUCT	ROLLBACK	STATIC_PREPARE_STEP
EVALUATE	STATIC_CHECK_INTEGRATION_STEP	STATIC_ROLLBACK

Some of the methods are as follows:

EVALUATE, PREDICT methods:

```
FUNCTIONS { o1 = fn_o1(i1,i2), PREDICT=fn_o1_predict( i1,i2 );}
FUNCTIONS { o1,o2 = fn_o1o2(i1,i2), PREDICT =fn_o1o2_predict(i1,i2);}
```

RESIDUAL method:

```
FUNCTIONS { o1 = RESIDUAL residual_fn_o1( i1,i2,o1 ) ; }
FUNCTIONS { o1,o2 = RESIDUAL residual_fn_o1o2(i1,i2,o1,o2 ) ; }
```

Note that residual methods must specify the target ports (left-hand side) as arguments to the callback to ensure correct dependency during the graph-theoretical analysis in *setupcpp*. Also, you can specify either an EVALUATE method or a RESIDUAL method, but not both.

DEFAULT_RESIDUAL method:

```
FUNCTIONS { o1 = fn_o1( i1,i2 ) ;
DEFAULT_RESIDUAL = fn_default_residual(o1,i1,i2) ; }
```

The default residual method lets you specify an inverse that will be used if no matching can be obtained with the explicit inverses. Also, a default residual method must list all the ports defined in the atomic class as arguments.

CONSTRUCT method:
DESTRUCT method:
PREPARE_STEP method:
CHECK_INTEGRATION_STEP method:
COMMIT method:
ROLLBACK method:
STATIC_CONSTRUCT method:

`STATIC_DESTRUCT` method:
`STATIC_PREPARE_STEP` method:
`STATIC_CHECK_INTEGRATION_STEP` method:
`STATIC_COMMIT` method:
`STATIC_ROLLBACK` method:

Example of static methods:

```
FUNCTIONS {
      x = fn_x( x, y )
      STATIC_PREPARE_STEP = fn_static_prepare_step()
      STATIC_CONSTRUCT =fn_static_construct()
      STATIC_DESTRUCT = fn_static_destruct();
}
```

Note that static methods have no arguments.

Example of an atomic class with multiple single-valued inverses:

```
FUNCTIONS {
   port1 = explicit_fun1( port2,port3,... ) ;

   port2 = explicit_fun2( port2,port3,... )
   PREDICT = predictor_fun2( port1,port2,port3,...)
   CONSTRUCT = construct_fun2( port1,...) ;

   port3 ;
}
```

Example of an atomic class with a multi-valued inverse:

```
FUNCTIONS {
   port5,port6 = explicit_multiOutFun1( port1,port2,port3,port4 ) ;
}
```

Note that there can only one multi-valued inverse per atomic class, thus making such a class a directed class.

Here is the example explained in detail:

```
FUNCTIONS {
   port1 = explicit_fun1( port2,port3,... ) ;
```

Here `explicit_fun1` is the *C++* function that calculates the value of `port1` using the values of `port2, port3,....`

```
   port2 = explicit_fun2( port2,port3,... )
   PREDICT= predictor_fun2( port1,port2,port3,...)
   CONSTRUCT= construct_fun2( port1,...) ;
```

Here `explicit_fun2` is the c++ function that calculates the value of `port2` using the values of `port2, port3,....` The `PREDICT`= is used to specify the predictor *C++*function (i.e. `predictor_fun2( port1,port2,port3,...)`) that calculates the predictor value of the integrators. The `CONSTRUCT`= is used to specify the constructor method function (i.e. `port1` for `construct_fun2( port1,...)`) that is used if '`port2`' is matched with the explicit function (i.e. `explicit_fun2( port2,port3,... )`) for this class by the *setupcpp* program.

```
   port3 ;
```

If there is no explicit *C++*function available for a port, either that port is not mentioned, or only the name of the port specified with the terminating semicolon in the `FUNCTIONS` statement. Note that if there is no explicit *C++*specified for a port, the method functions should not be specified.

```
   port5,port6,... = explicit_multiOutFun1( port1,port2,... ) ;
                }
```

Here, the function `explicit_multiOutFun1(..)` has the arguments `port1,port2,...` and computes the values of `port5,port6,...` .

REFERENCES

Anderson, J. L. (1986). *A Network Language for Definition and Solution of Simulation Problems*, Lawrence Berkeley Laboratory.

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

Brenan, K. E., S. L. Campbell, and L. R. Petzold, *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*, Elsevier Science Publishing Co., Inc., 1989.

Buhl, W. F., A. E. Erdem, et al. (1993). "Recent Improvements in *SPARK*: Strong Component Decomposition, Multivalued Objects, and Graphical Interface." *Proceedings of Building Simulation '93*, Adelaide, International Building Performance Simulation Association. Available from Soc. for Computer Simulation International, San Diego, CA.

Char, B. W., K. O. Geddes, et al. (1985). *First leaves: a tutorial introduction to Maple, in Maple User's Guide*. Waterloo, Ontario, WATCOM Publications Ltd.

Conte, S. D. and C. de Boor (1985). *Elementary Numerical Analysis: An Algorithmic Approach*. McGraw-Hill Publishing Co.

Dennis, J. E. and Schnabel, R. B. (1996). *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Classics in Applied Mathematics 16, SIAM.

LBL (1984). *DOE-2 Reference Manual*, Lawrence Berkeley Laboratory.

McHugh, J. (1990). *Algorithmic Graph Theory*. Englewood Cliffs NJ 07632, Prentice Hall.

Nataf, J.-M. and F. C. Winkelmann (1994*). Symbolic Modeling in Building Energy Simulation*. Energy and Buildings 21 (1994) 147-153.

Nierstrasz, O. (1989). "Survey of Object-Oriented Concepts." *Object-Oriented Concepts, Databases, and Applications*. W. Kim and F. H. Lochovsky. New York/Reading, ACM Press/Addison-Wesley**:** 3-21.

Nowak, U. and L. Weimann, *A Family of Newton Codes for Systems of Highly Nonlinear Equations*, Konrad-Zuse-Zentrum fuer Informationstechnik Berlin, Technical Report TR-91-10 (December 1991)

Press, W. H., B. P. Flannery, et al. (1988). *Numerical Recipes in C*. Cambridge, Cambridge University Press.

Rand, R. H. (1984). *Computer Algebra in Applied Mathematics: An Introduction to MACSYMA*. Boston.

Sahlin, P. and E. F. Sowell (1989). "A Neutral Format for Building Simulation Models." *Proceedings of Building Simulation '89*, Vancouver, BC, International Building Performance Simulation Association.

Sowell, E. F. (2003). "Extension of the SPARK kernel." *Proceedings of the IBPSA Conference*, Rotterdam, Netherlands.

Sowell, E. F. and W. F. Buhl (1988). "Dynamic Extension of the Simulation Problem Analysis Kernel (SPANK)." *Proceedings of the USER-1 Building Simulation Conference*, Ostend, Belgium, Soc. for Computer Simulation International.

Sowell, E. F., K. Taghavi, et al. (1984). "Generation of Building Energy System Models." ASHRAE Trans. **90**(Pt. 1): 573-86.

APPENDIX A: CLASSES IN THE *GLOBALCLASS* DIRECTORY

Classes in the *SPARK* globalclass directory represent general objects that can be applied to a wide range of problems. Each class has internal documentation in the form of a commented header. You should consult the header before using one of these classes. The classes are listed in Table A..

Table A.1: *SPARK* Global Classes

Class	Description
abm4	Adams-Bashforth-Moulton integration scheme of order 4
abs1	Absolute value
bd4	Backward Differences formula of order 4
bfd	Backward-Forward Difference formula of order 2
bound	Bound a value
clipnorm	Bound a value between 0 and 1
diff	Difference
equal	Equality
euler	Explicit Euler integration scheme of order 1
implicit_euler	Implicit Euler integration scheme of order 1
integrator_euler	Euler PC integration scheme of order 1 with error control.
integrator_trapezoidal	Trapezoidal PC integration scheme of order 2 with error control.
lin	Linear relation $A \cdot x_1 + B \cdot x_2 - C \cdot x_3 = 0$
lintrp	Linear interpolation
lintrp1	Linear interpolation to 1
log	Natural log
log10	Log base-10
max2	Larger of 2 values
min2	Smaller of 2 values
neg	Negation
polyn3	3rd degree polynomial
poslim	Force to be positive
pow	Exponentiation operator
propdiff	Point-slope equation of straight line
safprod	Safe product
safquot	Safe quotient

safrecip	Safe reciprocal
select	Logical if-then-else construct
square	Square of a value
sum	Sum of 2 values

APPENDIX B: USING THE HVAC TOOLKIT

THE *SPARK* HVAC TOOLKIT

The *SPARK* HVAC Toolkit is based on the ASHRAE Secondary Systems Toolkit (Brandemuehl 1993), supplemented with primary equipment models from the DOE-2 building energy simulation program (LBL 1984). This library of HVAC components is limited to steady state models. The models included are listed in Table B.1.

These classes are located in the *SPARK* hvactk/class directory. ach class has internal documentation in the form of a commented header. You should consult this header before using one of these classes.

Many of these classes are lower-level macro or atomic classes from which the user-level classes are built. These are automatically introduced into your problem as needed when you declare an object of the higher level class.

EXAMPLE USAGE

Some examples of using these classes have already been seen in examples in this manual. For example, we used the **cond** class in the *room_fc* problem in Section 6.5. In addition, every class has a test driver .pr file and associated .inp file in compressed form in pr.exe in the *SPARK* bin directory. You can access one of these test drivers by executing pr.exe with the class name as an argument, e.g.,

```
pr cond.pr <enter>
pr cond.inp <enter>
```

This will place the driver problem and input files for cond.cc in the working directory. Alternatively, you can execute the provided command file called testhvac to extract, build, and execute the driver. First, you should go to the *SPARK* hvactk directory. Then type:

```
testhvac cond <enter>
```

Results can be found in cond.out.

Note that the system models provided with the library show relatively complex macro classes that have been constructed from other Toolkit classes. These also have test drivers in the pr.exe compressed file.

Table B.1 *SPARK* HVAC Toolkit Classes

Class	Description
acdx	Direct expansion air-conditioning unit
airhx	Air to air heat exchanger
balance	Transport balance equation
bf	Coil bypass ratio relationship
bf_adp	Bypass factor/apparatus dew point coil
bf_ntu	Coil bypass factor vs. an Ntu-like parameter
boiler	Boiler
cap_rate	Moist air capacitance rate
capratel	Capacitance rate for water
cchiller	DOE-2 single-stage compression chiller
cclogic	Dry *vs.* wet-coil decision logic
ccsim	Simple cooling coil
cond	Generic conductance relation
cpair	Specific heat of air
ctfunc	Cooling tower model correlation
ctr1	Cooling tower Fr vs. range dependency
ctr2	Cooling tower Fr vs. approach dependency
cvrhsys	Constant volume reheat system
ddhtbal	Dual-duct zone convergence enhancer
ddsys	Dual-duct system
dewp_hw	Dew point using Hyland & Wexler saturation correlation
dewpt	Dew point relationship for moist air using Walton's saturation correlation
divsim	Diverter (splits a flow stream into two streams)
drcc1u	Dry-coil, cross flow, stream one unmixed
drccbm	Dry-coil, cross flow, both streams mixed
drccbu	Dry-coil, cross flow, both streams unmixed
drcctr	Dry-coil, counter flow
drcprl	Dry-coil, parallel flow
drywet	Dry/wet cooling coil
dxcap_m	Capacity variation with mass for DX AC unit
dxcap_t	DX AC unit capacity variation with outside dry and inside wet-bulb temperatures
dxeir_m	EIR variation with mass flow rate
dxeir_t	DX AC unit EIR variation with TWb

econ	Economizer
effc1u	Ntu-effectiveness, stream 1 unmixed
effcbm	Ntu-effectiveness, cross flow both mixed
effcbu	Ntu-effectiveness, cross flow both unmixed
effctr	Ntu-effectiveness for counter flow
effncy	Forces two *input*s to *sum* to 1.0
effntu1	Exponential effectiveness vs. Ntu
effprl	Ntu-effectiveness for parallel flow
eintrp1	Exponential interpolation
eir1_oc	Curve fit for eir1 in open centrifugal compressor
eir2_oc	Curve fit for eir2 in DOE-2 open centrifugal compressor
enthalpy	Enthalpy, dry-bulb, humidity relation.
enthsat	Dry-bulb vs. enthalpy at saturation
enthvap	Enthalpy of water vapor
enthwat	Enthalpy of water
enthxc1u	Enthalpy exchanger, cross flow, one stream unmixed
enthxcbm	Enthalpy exchanger, cross flow, both streams mixed
enthxcbu	Enthalpy exchanger, cross flow, both streams unmixed
enthxctr	Enthalpy exchanger, counter flow
enthxprl	Enthalpy exchanger, parallel flow
eq31	Equation 31 of 1993 ASHRAE Handbook of Fundamentals, Ch. 6
evaphum	Evaporative humidifier/cooler
fan_dd	Discharge damper fan, volume flow-temperature interface
fan_iv	Inlet-vane-controlled fan, volume flow-temperature interface
fan_vsd	Variable-speed-drive fan, volume flow-temperature interface
fann_dd	Discharge damper fan, mass flow-enthalpy interface
fann_iv	Inlet-vane-controlled fan, mass flow-enthalpy interface
fann_vsd	Variable-speed-drive fan, mass flow-enthalpy interface
fansim	Simple fan with part-load coefficients in the interface
fansim_n	Simple fan with part-load coefficient and enthalpy/mass interface
fflp_blr	Boiler part-load curve fit
fflp_dd	Fraction of full-load power for discharge damper fan
fflp_iv	Fraction of full-load power for inlet vane fan
fflp_vsd	Fraction of full-load power for variable speed drive fan
gendiv	Generic diverter
htxc1u	Cross flow, stream one unmixed heat exchanger

htxcbm	Cross flow, both streams mixed heat exchanger
htxcbu	Cross flow, both streams unmixed heat exchanger
htxctr	Counter flow heat exchanger
htxeff	Heat exchanger effectiveness
htxprl	Parallel flow heat exchanger
htxtemp	Temperature vs. capacity flow vs. effectiveness
humeff	Humidity exchanger effectiveness
humex	Humidity exchanger
humratio	Humidity ratio vs. partial pressure of vapor
idealgas	Ideal gas law
indep_fr	Independent fractions
indevap	Indirect evaporative cooler
lat_rate	Latent heat rate
mixer	Mixing box model for moist air
propcont	Proportional controller
pumpsim	Simple pump
rcap_oc	Curve fit for capacity in open centrifugal compressor
relh_hw	Relative humidity (Hyland & Wexler)
relhum	Relative humidity
rho	Moist air density vs. specific volume and humidity ratio
rhomoist	Moist air density vs. dry-bulb and humidity ratio
room	Simple room with heat loss and air mass
satp_hw	Saturated Pressure (Hyland & Wexler)
satp_r	Saturated pressure of water vapor, residual method
satpress	Saturated pressure relationship for water
sercond	Conductors in Series
specvol	Specific volume of air
tower	Cooling tower
tstdhb	Test driver for **ddhtbal**
varmix	Variable mixing box
vavsys	VAV System
vlvcirc	Flow circuit with non-linear valve and series flow resistance
wcoilout	Wet-coil leaving conditions
wetb_hw	Wet-bulb temperature (Hyland & Wexler)
wetbulb	Wet-bulb temperature
wtcc1u	Wet cooling/dehumidification coil, cross flow, one stream unmixed

wtccbm	Wet cooling/dehumidification coil, cross flow, both streams mixed
wtccbu	Wet cooling/dehumidification coil, cross flow, both streams unmixed
wtcctr	Wet cooling/dehumidification coil, counter flow
wtcprl	Wet cooling/dehumidification coil, parallel flow
zone	Simple steady-state thermal zone
zone_dd	Dual-duct controlled zone

APPENDIX C: PREFERENCE FILE FORMAT

WHAT ARE PREFERENCE FILES?

Preferences file are external representations of objects of class `TPrefList`. This *C++* class is designed to allow storage and retrieval of (`key`, `value`) pairs, somewhat like a mapping. However, this class differs from a typical mapping in that it allows an hierarchical description of information. The example below will allow you to better understand the structure and format of *SPARK* preference files.

USES OF PREFERENCE FILES IN *SPARK*

Preference files are used several places in *SPARK* to store information about important aspects of the problem and how it is to be solved. For example, every *SPARK* problem has a probName.prf file that gives information about the problem component structure, and how each component is to be solved (see Section 11). Also, each problem has a run-control file probName.run (see Section 18) with information about the simulation interval and other control issues. In some environments, a global spark.prf stores critical information about the *SPARK* installation. Here we explain the general format of all preference files.

HIERARCHICAL DATA: THE STRUCTURE OF THE PREFERENCE FILE

As an analogue of the way *SPARK* preference files are structured, consider how the description of a building might be stored. The building is to have a *Name*, a *Roof*, a *Floor*, and an arbitrary number of *Walls*. Although the *Name* has a simple string value, e.g., "MyBldg", *Roof*, *Floor* and every *Wall* is to have two attributes, *U* and *W*.

Figure C. shows this information as a general tree. It can also be thought of as an object called *theBuilding*. Every node in this tree can be viewed as a *key*, and the list of child nodes can be viewed as the *value* of that key. Thus *theBuilding* has a value which is the list *(Name, Roof, Walls, Floor)*, each of which is another tree. In turn, the root of each of these trees can be thought of as another key with its own value. The key *Name* has a single value, *myBldg*, and the key *Roof* has the value which is the list (*U*, *W*), each of which is a tree. The *U* and *W* keys at the roots of these trees each have a single value, (1.2) and (1.0) respectively. Note that nodes in the tree like *myBldg*, *1.2*, and *1.0* are distinctly different from nodes like *Name* or *Root* in that they have no children, i.e., they are leaves. Another way of saying this is that the "value" of a node like *myBldg* or *U* consist of an empty list ( ). These are the actual data stored in the structure. Note also that the path from the root to any leaf is a unique identifier of the data in the leaf. For example, *theBuilding.Roof.U* identifies the value 1.2.

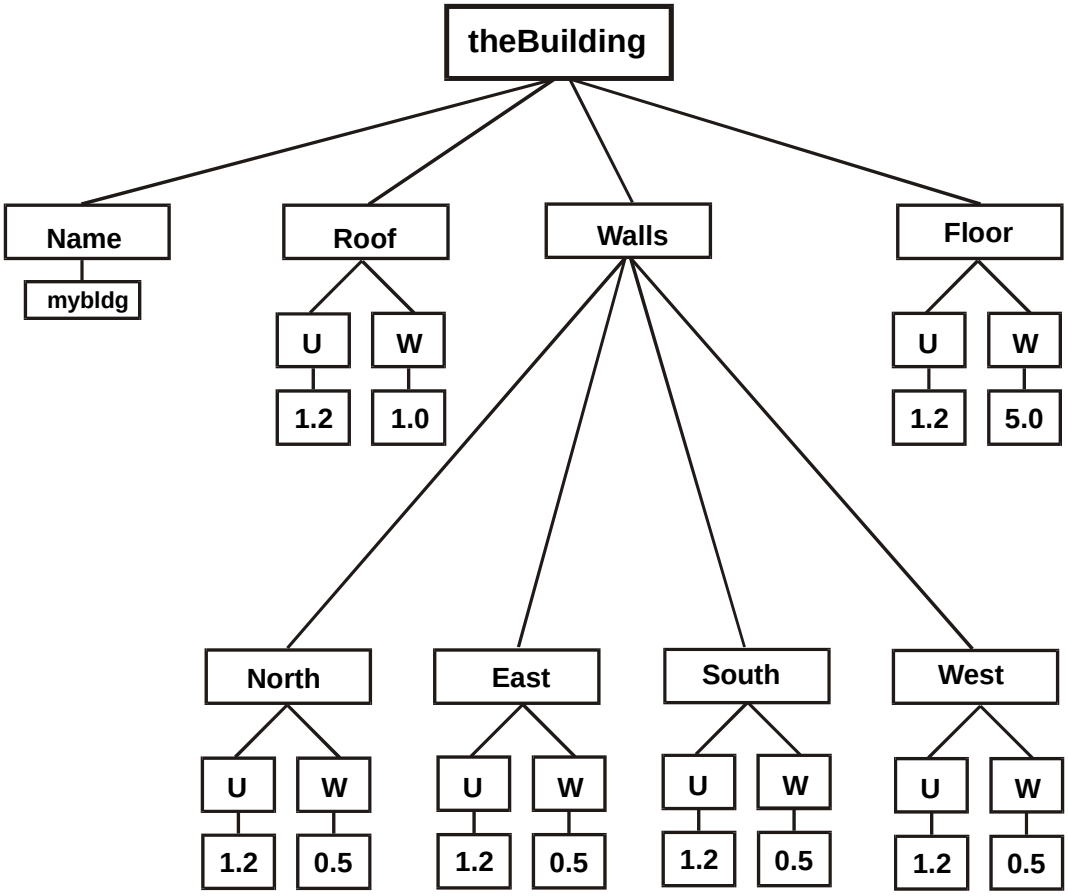

Figure C.1: Simple Building Represented as a Tree

PREFERENCE FILE FOR THE BUILDING DESCRIPTION EXAMPLE

The preference file expresses this tree structure as text. The preference file code example for the tree in Figure C. is shown below.

The format follows the convention that a key is followed by a list representing its value, enclosed in parentheses. If the list is empty, indicated by empty parentheses, the implication is that the key is in fact actual data. Note that the key representing the file itself, in this case *theBuilding*, is not part of the stored data. This is because externally the operating system will know it by the assigned file name, and programs that read preference files assign the file contents, i.e., its value, to an instance of *prefItem* class. Consequently, it is not useful to store the name in the file itself, and the file content begins with an opening parenthesis, and ends with a closing parenthesis. With these conventions, here is the file for *theBuilding*:

```
(
   Name (myBldg ( )
   )
   Roof (
      U (1.2 ( )
      )
      W (1.0  ( )
      )
   )
   Walls (
      North (
```

```
         U (1.2 ( )
         )
         W (0.5 ( )
         )
      )
      South (
         U (1.2 ( )
         )
         W (0.5 ( )
         )
      )
      East (
         U (1.2 ( )
         )
         W (0.5 ( )
         )
      )
      West (
         U (1.2 ( )
         )
         W (0.5 ( )
         )
      )
   )
   Floor (
      U (1.2 ( )
      )
      W (5.0  ( )
      )
   )
)
```

Since *theBuilding* tree has four first-level nodes, between file opening and closing parenthesis there are four main clauses, each consisting of a key followed by a parenthetic expression representing the value of the key. The first-level keys are the nodes in the tree, *Name*, *Roof*, *Walls*, and *Floor*. The *Name* key has a simple value, the building name string "myBldg", so it is followed by a empty parentheses. Note that the format is delimited entirely by the parentheses so spaces in strings are allowed, and no quoting is necessary. The *Roof* and *Floor* keys have values that are trees with nodes representing *U* and *W*. The *U* and *W* keys have simple values, so they are followed by empty parentheses. The *Walls* identifier has a more complex structure, namely four trees, each with a structure like *Roof* and *Floor*.

EDITING THE PREFERENCE FILE

Each *SPARK* problem has an associated preference file that sets important information needed by the solver. This file describes the settings for the numerical solution of each component of the problem . In addition, this preference file includes a list of the *C++* source files that are specific to the problem. As explained earlier, the problem-preference file, probName.prf, is generated by the *SPARK setupcpp* program at the same time that it generates probName.cpp. The following preference file is for the example.pr problem:

```
(
GlobalSettings (
   Tolerance (1.E-6 ())
   MaxTolerance (1.E-3 ())
)
ComponentSettings (
   0 (
```

```
      ComponentSolvingMethod ( 0 ())
      TrueJacobianEvalStep ( 1 ())
      Epsilon (1.E-6 ())
      RelaxationCoefficient ( 1.0 ())
      ScalingMethod ( 0 ())
      MaxIterations (50 ())
      MatrixSolvingMethod ( 0 ())
      PivotingMethod ( 1 ())
      RefinementMethod ( 0 ())
   )
)
Sources (
   ./example.cpp ()
   ../class/r1.cc ()
   ../class/r2.cc ()
   ../class/r3.cc ()
   ../class/r4.cc ()
)
)
```

Since probName.prf is a text file, any text editor can be used to edit it. Alternatively, you can use tools provided with *SPARK*. One of these tools is a command line program called *repref*. In general, execution of *repref* is as follows:

```
repref file.prf [pref 0] [pref n-1] action key  <enter>
```

This modifies the branches `[pref 0]` ... `[pref n-1]` according to `action`:

- `= key` replaces value at the branch by `key`
- `- key` removes value `key` value at the branch
- `+ key` add value `key` at the branch

As an example, to change `Epsilon` for Component 0 in example.prf:

```
repref example.prf ComponentSettings 0 Epsilon = 1.e-8  <enter>
```

Repref is handy for writing script files for preference file modifications.

APPENDIX D: *SPARK* PROBLEM DRIVER

The *SPARK* problem driver Application Programming Interface (API) allows an advanced user to implement a customized driver function in order to:

- customize the sequence of operations to re-solve the same problem,
- manage and solve multiple problems,
- retrieve solution values and specify new input values, and
- change run-control parameters between successive simulation runs.

The sparksolver.cpp file that implements the default *SPARK* driver function uses this set of API functions to carry out the simulation task.

Users can write a customized problem driver to retrieve the solution value from any problem variable and to modify the values of the input variables, so that multiple runs of the same problem can be carried out with different boundary conditions. Variables comprised in each problem can be looked up by names and by handle from the methods of the `TProblem` class. A variable handle is its unique identifier specified as an unsigned integer in the problem description files probName.cpp. and probName.xml (See Figure 1-1).

The problem driver API enables:

- the management of multiple problems in the same driver function as well as
- the integration of a *SPARK* problem within another program.

Comprehensive documentation on how to write a problem driver function can be found in the htm/chm tutorial *SPARK Build Process and Problem Driver API* that can be found in the *SPARK* doc directory and at http://SimulationResearch.lbl.gov in the *SPARK* area.

GLOSSARY OF TERMS

absolute tolerance (ATOL)
Absolute tolerance value.

algorithmic programming
A sequence of operations and assignments leading from prescribed inputs to prescribed outputs.

assignment
In computer languages, assignment is the action whereby a value is associated with an identifier representing a variable. Although the symbol "=" is often used for assignment, e.g., X = 2*y, assignment is different from mathematical equality because the latter implies that the expressions at the left and right of the "=" symbol are always equal. In particular, a sequence of assignments are order dependent, while a set of mathematical equations are not. See "algorithmic programming."

atomic class
A model comprising a single equation with used variables linked to its ports. Acts as a template for instantiation of atomic objects.

break level
An integer from 0 to 10 expressing the desirability of using the associated link to break cycles in the computation graph.

class
A general description of an equation (atomic class) or group of related equations (macro class). A class acts as a template for instantiation of objects.

command file
A file containing *MSDOS* commands. Also called a "batch" file .

continuous variable
Variable that can take on any real value between a minimum and maximum value.

cut set
A set of variables (links) that will break all cycles in the computation graph. *SPARK* attempts to minimize the size of the cut set. The variables in the cut set are called "break variables" and are used for iterative solution.

cyclic
In graph theory, the property of having closed paths, or circuits.

differential algebraic equation system
A system of differential and algebraic equations for simultaneous solution.

discrete state variable
A variable that can take on only specific values rather than any real value within a range.

dynamic variable
A variable for which the derivative appears in a differential equation.

environment variable
A symbol whose value is assigned in your computing environment, as opposed to within the *SPARK* program system. See documentation for Microsoft *Windows* for more information and to learn how environment variables are set. See also sparkenv.

GNU
GNU is not UNIX; GNU is a system of free software programs developed through the Free Software Foundation.

graph
See "mathematical graphs."

HVAC
Heating, ventilation, and air-conditioning.

ill-posed
A problem that is not well-posed is said to be ill-posed. See "Well-posed."

implicit inverse
A form of an equation in which a particular variable occurs on both the left and right sides of the equation. Used when explicit inverses cannot be obtained. Solution requires iteration.

initialization
Specifies the value of variable at `InitialTime`. Required for dynamic variables and break variables.

initial time
The time when the simulation starts. This is the time at which initial conditions for differential equations apply.

input set
The complete set of information needed to define execution of a *SPARK* problem. Includes input data files and run control information.

input/output free
A style of model expression that provides a set of equations rather than an algorithm. Since any set of inputs that leads to a well-posed problem can be specified in conjunction with these equations, it is sometimes called "input/output free."

instantiate
To create an instance of a class. To create an object based on a class definition. The `DECLARE` statement performs instantiation in *SPARK*.

integration formula
A formula used in numerical solution of differential equations to calculate a value for the integration variable at the next point in time. The formula can be explicit, in which case the new value appears only on the left side of the equation, or implicit in which case the new derivative also appears on the right of the equation.

interface variable
A class variable that is to be visible from outside. Interface variables are defined with the `PORT` statement.

inverse
A form of an equation in which a particular variable is isolated on one side of the equation; i.e., a formula for a variable. The formula is obtained by symbolic manipulation of an equation for a particular variable in the equation. An explicit inverse has the wanted variable on the left side only, while an implicit inverse has that variable in the formula as well.

Jacobian
The square matrix of partial derivatives of residual equations with respect to the break variables in a strongly-connected component.

macro classes
A group of *SPARK* atomic or other macro classes linked together through their respective ports to form a subsystem model. A macro class can be used wherever an atomic class can be used.

make
A utility program that creates a program from its composite parts, in response to commands embedded in a makefile. *GNU make* is used for both the UNIX and Windows implementations of *SPARK*.

makefile
An input file for a *make* program. Contains various targets, their dependencies, and commands for building them.

match level
An integer from 0 to 10 expressing the desirability of matching the associated link variable with the associated object port, and therefore with the inverse for this object port.

mathematical graphs
A structure comprising a set of vertices (nodes) and edges (arcs) that connect them. Often used to model systems of interacting entities.

object-oriented
A methodology in which the model behavior and data are encapsulated in a programming entity comparable to the physical entity that it represents.

panel
A discernible region within a window on your computer screen.

parser
The program that interprets the *SPARK* files that describe the model as the first step toward solution. Builds the setup file.

PDF
A portable file format from Adobe Systems that retains page layout and graphics. You need a special program, called *Acrobat Reader*, to view a file in PDF format. This program is freely available on the Internet.

predictor
Value of a break variable at beginning of iterative solution. Defaults to value at previous time step if not specified with `PREDICT_FROM_LINK`.

prf file

A file that contains various component settings (also called preferences) needed for a program to run. In a sense, a generalization of command line options and environment variables.

propagation
Process by which *SPARK* infers certain `LINK` or `PORT` statement settings, e.g., `ATOL`, `INIT`, `MAX` and `MIN`, from settings at lower or higher levels.

relaxation coefficient
Multiplier, usually a fraction, on calculated correction that is applied in order to get new break variable values during iterative solution.

retained state
Value that needs to be saved between successive uses of an object. Currently, *SPARK* objects cannot retain state internally. However, values of link variables are retained for four previous time steps.

run-control
Data controlling the solution phase for a *SPARK* problem, e.g., start time, finish time, time increment, and list of input files and output files.

setupcpp
A program used in the process of building a *SPARK* problem. Processes the setup file produced by *parser*.

solver
The executable program that *SPARK* builds to solve a particular problem. Called probName.exe (Windows) or probName (UNIX). The library used by *SPARK* in constructing this executable is also sometimes referred to as the "solver."

sparkenv
A command file for setting up your environment for running *SPARK* at the command line.

spawn
To create a computational process in a computer.

strongly connected component or strong component
In graph theory, a maximal set of vertices and edges that allow any vertex to be reached from any other vertex. In *SPARK*, a strong component corresponds to a separately solvable sub-problem that *SPARK* automatically determines using graph theory. Sometimes called simply "Component."

symbolic manipulation
Operations on mathematical expressions in terms of contained symbols, as opposed to numerical evaluation. The goal is often to solve for a particular variable in terms of all others in the expression, i.e., to obtain an inverse. Often done with computer software, i.e., computer algebra.

target
A file or other object that can be created with one of the command sequences in a makefile.

tool bar
A row or column of icons, usually at the top of a window, that can be clicked to perform commonly needed tasks. The icons usually are pictorial, suggesting what the tool does. For example, the **Print** icon on many *VisualSPARK* windows looks like a laser printer.

updating
Setting the value of Previous-Value Variable to the value of a variable specified with `INPUT_FROM_LINK`. Occurs at beginning of time step, before solving the components.

well-posed
A problem is said to be well-posed if it admits at least one solution. One requirement is an equal number of equations (objects) and unknowns (links). There also must be a complete matching, i.e., a matching of each variable to a unique equation inverse. However, problems can meet these requirements and still not be well-posed. For example, the two curves $y = f(x)$ and $y = g(x)$ may not intersect, so there is no value of x that satisfies both equations.

INDEX