SmartFiles: An OO Approach to Data File Interoperability*

Matthew Haines
Computer Science Department, University of Wyoming
<haines@cs.uwyo.edu>

Piyush Mehrotra
ICASE, NASA Langley Research Center
<pm@icase.edu>

John Van Rosendale
ICASE, NASA Langley Research Center
<jvr@icase.edu>

Abstract

Data files for scientific and engineering codes typically consist of a series of raw data values whose description is buried in the programs that interact with these files. In this situation, making even minor changes in the file structure or sharing files between programs (interoperability) can only be done after careful examination of the data files and the I/O statements of the programs interacting with this file. In short, scientific data files lack self-description, and other self-describing data techniques are not always appropriate or useful for scientific data files. By applying an object-oriented methodology to data files, we can add the intelligence required to improve data interoperability and provide an elegant mechanism for supporting complex, evolving, or multidisciplinary applications, while still supporting legacy codes. As a result, scientists and engineers should be able to share datasets with far greater ease, simplifying multidisciplinary applications and greatly facilitating remote collaboration between scientists.

*This research was supported by the National Aeronautics and Space Administration under NASA Contract No. NAS1-19480 while the authors were in residence at ICASE, NASA Langley Research Center, Hampton, VA 23681. More information on SmartFiles can be found at `http://www.icase.edu/~haines/html/smart.html`.

1 Introduction

Data files play a fundamental role in scientific computing. While there are scientific programs that require no input and produce little output, most large scientific programs interact with a number of data files for both input and output. Moreover, scientific programs are often coupled via data files with other programs to create large systems. These systems provide the basis for multidisciplinary applications, such as those used for aircraft design and environmental simulation.

Data files are in many ways ideal for coupling programs and sharing information between researchers. They are persistent, they can be accessed from different languages, and they do not create problems with overlapping name spaces. Unfortunately, data files usually consist only of a series of bytes, whose syntax and semantics are implicitly defined by the programs that interact with them. In consequence, the "narrow" interface between programs that file I/O might otherwise provide is lost, leaving one with a very broad and unwieldy interface.

The basic problem with the current situation is that the structure and meaning of a data file is *implicit* in the programs interacting with it, the comments (if any) in the file, the directory in which the file is stored, the name of the file, and so on. Thus trying to use a file produced elsewhere is roughly analogous to the problem of interacting with external data structures without having access to the routines that create and manipulate the data structures. Object-oriented programming languages provide a solution to the latter problem, by coupling the data structures with the routines that manipulate them. We believe that the object-oriented methodology extends naturally to data files as well, yielding many of the same benefits to data files as it does in the context of programming languages.

Our approach to the software engineering issues inherent in data files is to replace current data files with "smart files," object-oriented analogs of current "dumb files." A smart file consists of a file descriptor, the data itself, and a set of associated library routines for interacting with the data at a relatively high level of abstraction. The access routines can also provide novel filtering capabilities, such as units conversion and consistency checks, not available with "dumb" files. Our goal is to apply the principles of encapsulation, modularity, and inheritance to data files, resulting in a cleaner file abstraction and greatly simplifying the interaction of users with complex scientific and engineering programs.

This paper describes the concepts of the **SmartFile** system. In section 2 we provide an overview of the system and its capabilities, and a comparison to related research is provided in Section 3. In section 4 we describe file types and the language used to define file types: DAFT (DAta File Types). Section 5 describes the interaction between SmartFiles and legacy systems; Section 6 describes the mechanisms for supporting automatic conversions; and Section 7 introduces the core access routines used to interact with the files. In section 8, we provide an example of a SmartFile for unstructured grids, and we provide current status and future directions in section 9.

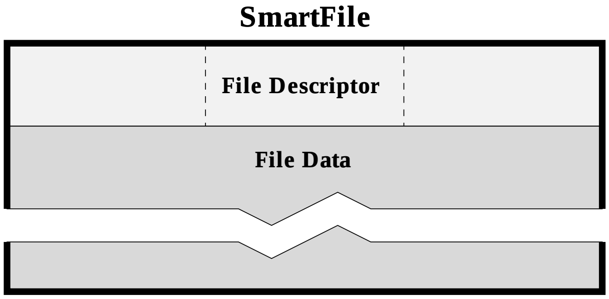

Figure 1: Logical view of a SmartFile

2 Overview

The notion of self-describing data implies that a SmartFile must at least contain both raw data and descriptive data (meta data in database terminology). Therefore, as depicted in Figure 1, a SmartFile consists of two main sections:

- a *file descriptor*, which describes the syntactic and semantic contents of this file; and
- a *file data section*, containing the actual data needed by application programs.

For various reasons related to performance and legacy system support, the physical definition of a SmartFile may or may not resemble this logical view of a SmartFile. However, it is important that from the user's perspective, both sections of the file are contained within a single Unix file, thus allowing the standard Unix commands (`cp, mv, tar, ci, ...`) to operate on SmartFiles. To do otherwise would mean providing separate interfaces for all Unix commands applicable to files.

Currently, the SmartFile descriptor consists of three components:

1. a *file type*, which can be a stand-alone type or a member of a complex hierarchy of file types;
2. a *layout*, which provides a detailed syntactic description of the organization of the raw data in relation to the abstractions (fields) defined by the file type; and
3. the *attributes*, which provide file-specific ancillary information (e.g. data, author, etc.) necessary for proper interpretation of the data file.

File types define a collection of *abstractions* or *fields* composed from data types, parameters, and attributes. The data types may be simple (e.g. `double`) or complex (e.g. `struct`), and arrays of any data type are supported. Parameters provide for generalized file types on the basis of variable-length fields, by allowing arrays to be defined in terms of an abstract parameter value rather than being restricted to fixed sizes. Attributes provide ancillary information about a field, file type, or file. Examples of common attributes include units of measure, date/time stamps, system of mapping used, etc. A detailed description of file types is given in Section 4.

The following example illustrates the specification of a field called `pressure`, defined as an array of `doubles` whose length is based on a parameter named `nnodes`, and whose units are specified in pounds per square inch:

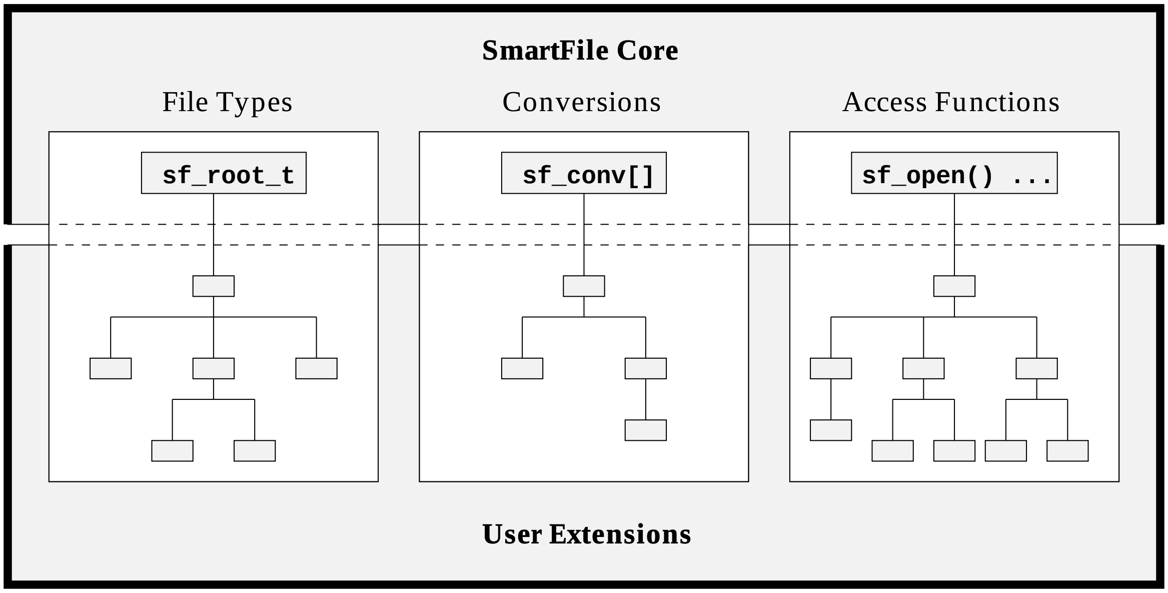

Figure 2: SmartFile system organization

```
field pressure[nnodes]: double <units=psi>;
```

In addition to providing descriptive information, attributes can be used to automatically drive conversion and filtering tools provided by the system or as user routines. From our example, if pressure is desired in millimeters of Mercury rather than pounds per square inch, then an automated conversion mechanism (`<mmHg> = <psi>*51.7151`) can be invoked to make the conversion on-the-fly. Section 6 provides more details on using attributes to trigger automatic conversions.

2.1 System Organization

The SmartFile system is organized into three extensible components (as depicted in Figure 2): file types, conversion tables, and access routines.

Each data file is an instantiation of a given file type, similar to the relationship between an object and a class. The file types are either provided by the core SmartFile system, provided by a domain-specific extension, or provided by the user. File types can be created stand-alone or from other file types, using inheritance to create a file type hierarchy. Section 4 details file types and the DAFT language used for creating them.

Conversion tables are used to direct the conversion of data elements from one form to another, or from one data structure representation to another. For units conversion, automated tables are available [11], and in the case of data structure conversions the user must provide the routines that are triggered by the attributes. Section 6 describes the details of conversion tables and how conversions are initiated.

The access routines provide the user with a simple interface with which to create and query a SmartFile. The interface can easily be supported in both C and Fortran, and users are free to extend the interface as needed by writing higher-level routines in terms of the given, low-level routines. Section 7 details the SmartFile access routines.

The extensible design of the SmartFile system allows application areas (such as computational fluid dynamics) to provide common file types, conversion tables, and access routines that can be shared among users of that community. Additionally, user's can extend the system to further refine its operation for their exact needs.

2.2 Capabilities

The current modus operandi for scientific programmers is to use the standard I/O routines provided through either Fortran or C to create data files with arbitrary representation. The goal of SmartFiles and other related systems is to elevate the programmer to a higher level of abstraction so that interaction with a file can be done at an abstract, conceptual level rather than at a raw, byte level. In the process of designing SmartFiles, we have identified the following capabilities as being essential towards achieving this goal:

- Interacting with file data using user-defined abstractions (fields).
- Specifying general file types that are related to data files as classes are related to objects.
- Providing the ability for the user to extend all portions of the design, including file types, conversion/filtering capabilities, and access/interface routines.
- Providing consistency checks to ensure that written data conforms to the file type specification.
- Providing an automatic filtering mechanism by combining attribute information with conversion tables/routines.
- Supporting legacy data files and programs. Since many scientific programmers are using legacy programs or interoperating with people who are, support for legacy files and systems is tantamount to interoperability.

SmartFiles conforms to Booch's specification of an object-oriented system [3] by supporting *abstraction*, *encapsulation*, *modularity*, and *hierarchy*. Abstraction and encapsulation are supported by interfacing with the file using access routines that are driven by the file type information. Thus, a software layer is placed between the user and the data file, allowing for filtering, consistency checks, and performance optimizations. Modularity is supported in the ability to decouple the segments of a SmartFile for support of legacy systems. Hierarchy (inheritance) is supported by allowing the user to specialize file types, conversion tables and routines, and access routines from general abstractions.

3 Related Research

The notion of adding syntactic and semantic descriptions to files is not new: Pablo [2] utilizes a self-describing data format for performance trace files; netCDF [9] provides a self-describing format for multi-dimensional tabular data (such as sensor data); and HDF [8] is a self-defining file format for transfer of various types of data (n-dimensional data, raster images) between different machines. These systems all impose a specific structure on the data that can be represented (such as multidimensional tables), in a sense restricting their users to a single file type whose field names and values may vary. SmartFiles allows for true user-defined file types, where the user is in complete control over both the structure and content of the data fields. Thus a representation for a graph structure can be done as easily as a multidimensional table. This is crucial for many engineering programs, such as adaptive mesh refinement in computational fluid dynamics, which use data files to store grid configurations. Additionally, SmartFile data types can be generalized for a wide variety of similar data files by incorporating runtime values called *parameters* into their definition. This allows for the same field type to occur in different quantity for different data files and yet still belong to the same file type.

There are also a variety of standardized, application-specific data formats, including FITS [7] for astronomical data, GRIB [10] for meteorological data, PDS [6] for space mission data, SDTS [4] for geographical data, and a variety of graphical data formats, including TIFF, GIF, and JPEG. SmartFiles does not restrict the user to a specific format; rather, the user is allowed to create new formats (file types) from scratch or from existing file types through inheritance.

ELFS [5] describes an object-oriented approach to high-performance file I/O, in which files are treated as typed objects. The file objects encapsulate the details of obtaining high-performance I/O on a variety of parallel and distributed machines. While ELFS and SmartFiles both apply OO methodology to file access, the goals and implementations are different. ELFS strives for encapsulation to support high-performance I/O in a parallel or distributed system, whereas SmartFiles strives for creating files that are self-aware and can perform conversions and consistency checks. ELFS is implemented as an OO system in Mentat (C++), whereas SmartFiles supports a simple language to allow the user to specify the file type, or simply use an existing file type, and the interface is supported in both C and Fortran.

On the other side of the capabilities spectrum for self-describing data files are object-oriented databases. Although SmartFiles do support some OODBMS features, such as self-describing data and the ability for the end-user to create hierarchical data abstractions, they do not attempt supporting the more difficult (and resource consuming) functions such as concurrency control, automatic recovery, and complex query facilities [1]. By omitting these capabilities unnecessary for their intended use, we can create a much faster, easier to use, and compact tool.

Perhaps the single largest problem with many of the existing solutions to improving data files is the lack of support for legacy systems. We have provided support for integration of legacy files and programs into the system so that data file interoperability can occur even when legacy systems continue to use raw data formats.

4 File Types

In the object-oriented language C++, a `class` is used to provide a semantic description for a set of abstractions that will apply to a new data type. Objects, in C++, are then instantiations of this new data type with a given set of values for the abstractions. Even though the objects themselves may differ, it is always the case that two objects with the same data type, or class, will represent the same physical abstractions.

SmartFiles extends this idea to files by introducing a mechanism for creating and specifying file types that are to be associated with data files. Similar to data types, file types provide a semantic description for a set of abstractions, called *fields*, within a file. A SmartFile is then an instantiation of a given file type with a particular set of values for the fields. However, unlike a C++ class, the arrangement and storage of fields within a file type may differ from file to file of the same type. For example, two data files representing unstructured grids

```
filetype sf_point_t = {
    <type=sf_point_t>;
    parameter npoints;
    x, y : int <units=inches>;
    field point[npoints] : {x,y}
        <system=cartesian>;
}

*** SF_TYPE (sf_point_t) ***
*** SF_LAYOUT (              65) ***
5
0 1
1 2
2 3
3 4
4 5
*** SF_FIELD (npoints 37 1) ***
*** SF_FIELD (point 40 5) ***
```

Figure 3: Sample SmartFile

may both be of the same file type, but may have a different arrangement of the fields within the file type. To accommodate this flexibility in representation, a SmartFile descriptor (cf. Figure 1) consists of both a file type, which defines the field abstractions in the file, and a *layout*, which defines how the fields for the given file type are mapped to the physical storage of a file. Thus, SmartFile "objects" are instances of general file types coupled with a file-specific layout. Note, however, that file layouts are maintained automatically by the SmartFile access routines and, except for legacy systems (cf. Section 5), are transparent to the user.

Figure 3 depicts a sample SmartFile with its file type[*], layout, and data. SmartFiles may be physically stored in binary or ASCII form (in this case ASCII), and the user is given the option when the file is created.

The definition of a file type, specified using the DAFT specification language, consists of three types of declarations: attributes, parameters, and fields.

4.1 Attributes

Attributes provide a mechanism for associating ancillary, descriptive information relating to the elements of a SmartFile, including the fields, file types, and files themselves. Attributes consist of a pair of <*name, value*> strings, where the the value of the string is open for interpretation by the user. Most commonly, actual string values are used, such as `<units=cm>`, though numeric values are possible as well using the standard C routines `sprintf` and `atof/atoi`.

In addition to providing the user with general information about the file and its fields, attributes can be used as a triggering mechanism for implicit filtering of the data as its being read or written (cf. Section 6). For example,

[*] For illustration purposes, we have included the actual DAFT description for the `sf_point_t` file type, when in reality only the file type name would actually appear in the SmartFile descriptor.

using the udunits [11] library, attributes specifying units of physical measure can automatically and efficiently be converted as the data is being read or written. Although file attributes are not new, comments are a kind of unstructured attribute that have been the norm for scientific data files, using them as a triggering mechanism for implicit conversions is a useful and novel approach for scientific data files.

4.2 Parameters

Parameters provide a method for generalizing file types by providing a mechanism for specifying symbolic size and shape relationships for fields. Depending on when the symbolic parameter is *bound* to a value, called *binding time*, the effect can range from a static field size to a dynamic field size. This makes it possible, for example, to use the same file type given in Figure 3 for a file with 5 points as well as for a file with 5000 points, since the number of points is specified as a parameter. However, even though the files contain a different amount of information, they can both be considered to be of the same type and therefore the same abstractions are available to the user, in this case `points`. Since parameters are always used to indicate size, they must be a positive integral type, or in the case of our implementation, `unsigned int`s.

The binding time for parameters can potentially occur at any point from the specification of a file type to its use, but we have identified three times that seem to be the most useful:

1. *static*, where the parameter is bound to a static value in the file type description;
2. *early*, where the parameter is bound by the user (using a `sf_bind` call) before accessing any field based on that parameter; or
3. *late*, where the parameter is bound by the SmartFile system at file closing based on the number of items that were written for the fields based on that parameter.

While late binding provides flexibility and ease-of-use for a user, static and early binding provide the opportunity to provide coherence checks on the values used to create a field, ensuring that the number of elements "put" into a field match the pre-specified parameter value for that field. For late binding, the only coherence check possible is symbolic, such as that two fields based on the same parameter have the same number of elements each. Using parameters to obtain flexible file types and to provide automated coherency checks is another novel feature of SmartFiles.

4.3 Fields

Fields provide the mechanism that allows the user to define data abstractions that will be used for interacting with the file data, constituting the heart of the file type. Specifically, fields provide persistent, user-defined data structures, and in this way SmartFiles are similar to OODBMS. Fields may be defined as simple data types (eg., `int, float, double, ...`), or arbitrary structures recursively constructed from simple types.

For example, the `point` field defined in Figure 3 is a complex structure consisting of two integer subfields, `x` and `y`. A field is classified as a *subfield* if it is used to construct a larger field and lacks the "field" keyword,

```
filetype sf_point_t = {
    parameter npoints;
    x, y : int;
    field Point[npoints] : {x,y}
        <system=Cartesian>;
    field Length[npoints] : double
        <units=meters>;
}

filetype sf_my_point_t : sf_point_t = {

    field Line[npoints] : int;
    field Length[npoints] : double
        <units=feet>;
}
```

Figure 4: Example of file type derivation

meaning that it cannot be used as a data abstraction for the `get/put` primitives.

As an example of how fields are used to access data elements within a file, consider again the `point` field from Figure 3. A `point` element can be retrieved from the file using the `sf_get` primitive:

```
err = sf_get (sf, "point", &point_arr,
               &count, "system=Cartesian");
```

as opposed to the standard method:

```
for (i=0; i<count; ++i) {
    fscanf (fd, "%i%i",
            &point_arr[i].x,
            &point_arr[i].y);
}
```

4.4 Derived File Types

One of the most powerful features of any object-oriented system is hierarchy (inheritance): the ability to build new objects from existing objects. SmartFiles provides for file type hierarchies by allowing the user to derive new file types from existing file types. Under derivation, the abstractions defined in the base file type are available in the derived file type for extension or modification. All declarations in the base file type are present in the derived file type, unless redefined, and additional declarations may be made in the derived file type.

For example, in Figure 4 the file type `sf_my_point_t` is derived from the file type `sf_point_t`, where a new field, `Line`, is defined and the attribute value for an existing field, `Length`, is modified.

The inheritance mechanism for SmartFile file types forms an "is-a" relationship hierarchy, and any file can be opened using the type with which it was created or using any base type from which the actual file type was derived. This provides the ability for two scientists to interpret the same data file in different ways, which is necessary for supporting interoperability.

4.5 The DAFT Language

The DAFT (DAta File Type) language is used to declare the fields, parameters, and attributes that comprise a SmartFile file type. The syntax, shown in Figure 5, was chosen to be both simple and powerful, allowing attributes to be associated with subfields, fields, and file types. Parameters must be declared before their use, since a one-pass compiler will be used to parse the language, and the **field** keyword is applied to any data item which is to be *visible* to the user via the SmartFile access routines. Specification of the syntax in Figure 5 uses [item] to indicate an optional item and { item ...} to indicate zero or more occurrences of item.

The DAFT compiler takes the file type declarations and produces an enhanced symbol table that is used by the SmartFile access routines. The point at which the compilation takes place is still under investigation. Currently, a file type is associated with a SmartFile when the file is opened, and is specified in the form of a string that refers to a file containing the DAFT syntax. A search path similar to the Unix **BINPATH** is used to locate the DAFT file description. Once found, the specification is compiled (interpreted) and the results added to the global file type symbol table. We are also investigating the approach of pre-compiling the DAFT syntax and storing binary representations of symbol table entries that can quickly be added to the global table upon opening a SmartFile.

5 Legacy Systems

A shortfall of many existing systems for supporting improved access to data files is their inability to effectively integrate legacy data files and programs with the system. While creating a new data system out of whole cloth is attractive, this "all-or-nothing" approach would clearly hinder users who want to utilize the new features, yet still maintain compatibility with legacy programs and existing data files, or to interact with others in the community who are not using the new system.

In consequence, the SmartFile system was designed to allow interoperability with legacy users and systems. This is done by providing an easy and automated way of creating a SmartFile from a legacy file and vice versa.

To create a SmartFile from a legacy file, the user must provide three things: the legacy data file, a file type descriptor that contains the SmartFile abstractions, and a layout descriptor that maps the given abstractions to physical locations in the data file. These three elements are then passed to a special routine, **sf_pack**, which parses the file type and layout descriptors and creates a correctly-formatted SmartFile.

The layout descriptor is simply a series of declarations of the form <**fieldname:count**>, where the order of the declarations specifies the corresponding order for the data file. For example, consider the data for a plotting program that consists of some number of two-dimensional coordinate pairs preceded by the number of points in the file (as depicted in Figure 6). A possible file type descriptor might be:

```
filetype sf_point_t = {
```

```
filetype-desc    →  filetype id [ ‘:’ id ] ‘=’ ‘{’ spec-part ‘}’

spec-part        →  spec-item ‘;’ { spec-item ‘;’ ... }

spec-item        →  param-decl
                 |  attr-list
                 |  struct-decl
                 |  field-decl
                 |  subfield-decl

param-decl       →  parameter param-item { ‘,’ param-item ...}

param-item       →  param-id [ ‘=’ integer-const ]

attr-list        →  ‘<’ attr-item { ‘,’ attr-item ...} ‘>’

attr-item        →  attr-id ‘=’ attr-value

struct-decl      →  struct struct-id struct-def attr-list

struct-def       →  ‘{’ struct-item { ‘;’ struct-item ...} ‘}’

struct-item      →  subfield-id
                 |  subfield-decl

field-decl       →  field subfield-decl

subfield-decl    →  subfield-item { ‘,’ subfield-item ...} ‘:’ type attr-list

subfield-item    →  subfield-id { ‘[’ bounds-list ‘]’ }

bounds-list      →  bounds-item { ‘,’ bounds-item ...}

bounds-item      →  param-id
                 |  integer-const

type             →  scalar-type
                 |  struct-def

scalar-type      →  int
                 |  float
                 |  double
```

Figure 5: DAFT Syntax

```
5
0 1
1 2
2 3
3 4
4 5
```

Figure 6: Sample legacy file for plotting program

```
    parameter npoints;
    x,y: int;
    field point[npoints]: {x,y};
}
```

and the corresponding layout descriptor would then be:

```
npoints:1
point:npoints
```

The resulting SmartFile would then resemble the file in Figure 3. Note that the layout descriptor can also make use of the specified parameters, so that the same layout descriptor can be re-used when a new legacy data file needs to be re-packed. This is an important consideration because it allows the file type and layout descriptors to remain unchanged and still accommodate new versions (updated datasets) of the legacy data file.

To extract the legacy portion of a SmartFile, the **sf_unpack** routine is used to simply strip away the layout and file type information.

By providing a clean and un-cumbersome method for interoperating with legacy systems, SmartFiles offers scientific programmers the unique opportunity to gradually convert programs from standard language I/O primitives to SmartFile primitives while still remaining compatible with older datasets. Moreover, the user who has completely made the transition to the SmartFiles paradigm can still effectively cooperate with colleagues who continue to use raw data files. This is an important consideration that is often overlooked by many software system designers.

6 Data Conversion

One of the most common programming operations that scientific programmers undertake is writing filters to convert the output data from one program into a suitable format as input data for another program. Large scientific applications are often written as a collection of programs forming a "pipeline," with such filters at each step of the pipeline, and changes in one program can cause a ripple in the pipeline that necessitates changes in all of the subsequent filtering programs.

SmartFiles attempts to remedy this situation by firstly de-coupling the dependence that a program has on physical file formats, using data abstractions instead of actual I/O statements. Secondly, they provide a triggering mechanism (via the attributes) to apply implicit filtering techniques when data abstractions are being read or written. Since SmartFile data is not just passive ASCII data, but has attached "semantic content" via the file type and the attributes of the file, the system can perform whatever supported data conversions are requested during insertion or extraction of information. For example, if a file contains a field of distances in meters, but one prefers distances in feet, the *sf_get* command can perform this conversion by specifying the attribute <**units=feet**>. The SmartFile library notes the difference between the requested attribute and the stored attribute, and consults the conversion tables for a **meters** to **feet** equation, finding **<feet> = <meters>*3.28084**.

Units conversion is but one example of the kinds of conversions frequently needed with scientific data. The set of conversions envisioned for support include:

```
canonical units:
    cm                           % length
    kg                           % mass
    newton                       % force

conversion factors:
    1 inch =  2.54 cm            % length
    1 foot = 12 inch             % length
    1 barleycorn = 0.3333 inch   % length
    1 furlong = 660 foot         % length
    1 kg   = 1000 gram           % mass
    1 dyne =  1 gm cm / sec^2    % force
    1 erg  =  1 gm cm^2 / sec^2 % energy
```

Figure 7: Sample conversion table

1. Change of physical units (e.g. `furlong` to `barleycorn`[†]).
2. Change of coordinate systems (e.g. `polar` to `cartesian`).
3. Conversion between different *sets* of physical quantities (e.g. `momentum` and `density` to `velocity` and `pressure`).
4. Change of data structure representation (e.g. cell-centered nodes to vertex-centered nodes).
5. Interpolations and smoothing operations.

The first three of these conversions can be done with conversion tables, such as the one depicted in Figure 7. At runtime, when a conversion is needed, these tables are consulted to determine if there exists an appropriate path from the known value to the desired value. External conversions systems, such as Unidata's udunits package [11], can also be consulted. The set of equations defined by the conversion path is then applied to each data element as it is being streamed into or out of the system.

Table-driven interpretation has several advantages. For one, it is easy to *inherit* and *specialize* tables for extending the system to new disciplines. For example, an astronomer may need units like `solar-masses`, and `parsecs`, in addition to the units in the default tables, and these new units can easily be added. Also, table-driven interpretation is quite powerful. As a trivial example, compound units like $newton\text{-}cm^2/sec^2$ create no problem — as long as the system understands `newton`, `cm`, and `sec` separately, it can treat the whole as an algebraic composite of these simpler units. Similarly, given equations relating `density`, `energy`, `velocity`, `pressure`, and so on, the system can derive new equations to provide conversions not explicitly provided.

The last two conversions listed, change of data structures and the various smoothing and interpolation operations, require the user to provide additional code to perform the conversions. There is, for example, no automated method for converting from cell-centered nodes to vertex-centered nodes. However, while the user is required to provide such code, the system provides the triggering mechanism to automatically invoke these conversions depending on the way in which user's want the data to be retrieved/stored.

[†] A barleycorn is $\frac{1}{3}$ inch, as the astute reader must surely know.

7 Access Routines

The access routines provided by the SmartFile core library support opening and closing a file, getting and putting fields from a file, and inquiring and setting attributes. The routines support both C and Fortran interfaces, and we envision also supporting this interface from a C++ class. For simplicity, we detail the interface routines using the C interface.

```
int sf_open (const char *fileName,
             const char *fileType,
             sf_openmode_t openMode,
             sf_filemode_t fileMode,
             sf_file **smartFile);
```

- The `sf_open` routine attempts to open a SmartFile either for creation or as an existing file. The `fileType` is used to locate the corresponding descriptor file, "fileType.sfd," located in the file type directory path. The filetype descriptor contains the DAFT code that declares the available abstractions. The declarations are parsed and loaded into a symbol table associated with the active SmartFile descriptor defining this file. Additionally, the fields are "flattened" for optimal reading/writing. The openMode specifies whether the file should be written (create) or read, and the fileMode determine whether the file is ASCII or binary.

```
int sf_close (sf_file *smartFile);
```

- The `sf_close` routine closes a SmartFile and performs the integrity checks. There are two types of integrity checks: *completeness* and *coherence*. The first, completeness, determines if all fields exist in the file as specified by the file type. This check is important when the file will be used later, by another program that will expect certain information to exist. The second check, coherence, determines whether parameter-based fields have the appropriate number of elements based on their parameter values. All unbound parameters are also bound at closing time.

```
int sf_get (sf_file *smartFile,
            const char *fieldName,
            void *getLocation,
            char *desiredAttr,
            int *getCount);
```

- The `sf_get` routine retrieves the data stored in a given field of a SmartFile. The address where the data is to be stored is given by `getLocation`, and `getCount` is used to return the number of items read. To avoid problems with ordering of matrices, all field names must return either a scalar value or an array of values, such that they will be stored contiguously starting from `getLocation`. If `getCount` is specified as input, then only that many items will be read. If `getCount` is zero on input, then the entire field will be read. Successive reads from the same field will keep track of their place, so that the user can incrementally access the field data. This ability is needed to check for read-termination conditions that are based on the data being read – a common technique for scientific programming. The `desiredAttr` argument can be used to specify how the user expects the data to arrive, and is compared against the actual attributes for the specified field in the file type. If there is a mismatch, then the system searches for a conversion path in the conversion tables. If found, the appropriate conversions are made as the data is being read, otherwise an error is reported.

```
int sf_put (sf_file *smartFile,
            const char *fieldName,
            void *putLocation,
            char *desiredAttr,
            int *putCount);
```

- The `sf_put` routine writes the data that is stored starting at `putLocation` into the field specified by `fieldName`. As with `sf_get`, successive puts to the same field will keep track of their place, and `putCount` is used to specify how many field values are to be written. The `desiredAttr` argument can be used to specify information about the data to be placed into the specified field so that the system can apply conversions as necessary.

```
int sf_bind (sf_file *smartFile,
             const char *parameterName,
             unsigned int value);
```

- The `sf_bind` routine allows the user to bind a value to a parameter that is not statically-bound by the file type, so that consistency checks performed when the file is closed can be more accurate. Of the three binding times mentioned in Section 4.2, `sf_bind` performs *early* binding.

```
int sf_pack (const char *fileType,
             const char *rawDataFile,
             const char *layoutDescFile,
             const char *smartFile);
```

- The `sf_pack` routine creates a SmartFile from a raw data file, file type, and layout descriptor as explained in Section 5.

```
int sf_unpack (const char *smartFile,
               const char *rawDataFile);
```

- The `sf_unpack` routine creates a raw data file from a SmartFile by simply stripping away the SmartFile components of the file.

```
int sf_inqattr (sf_file *smartFile,
                const char *fieldName,
                const char *attrName,
                const char **attrValue);
```

```
filetype Unstructured = {
    <type=Unstructured>
    parameter nnodes, nedges, ncells,
        nsedges, nvedges, nfedges;
    node1, node2, cell1, cell2 : int;
    field edgelist[nedges] :
        {node1, node2, cell1, cell2};
    field slist[nsedges] : int;
    field vlist[nvedges] : int;
    field flist[nfedges] : int;
    x,y : double;
    field coords[nnodes] : {x,y}
        <system=cartesian>;
}
```

Figure 8: Example of unstructured grid file type

- The `sf_inqattr` routine retrieves the attribute information for a given field or file (if field is `NULL`). The name of the attribute is specified, and a pointer to a string containing the value of the attribute is returned.

```
int sf_addattr (sf_file *smartFile,
                const char *fieldName,
                const char *attrName,
                const char *attrValue);
```

- The `sf_addattr` routine allows the user to assign the given value to the attribute specified for the field or file (if field is `NULL`).

8 Example

In this section, we give a brief example of a SmartFile and its use. Figure 8 depicts the file type descriptor for an "unstructured grid" file type, useful in computational fluid dynamics. The fields in this file type include: an *edgelist*, describing the mesh topology (essentially a planar graph), a *coords* list giving the location of the mesh points, and lists describing edges on solid, viscous, and far-field boundaries. Parameters control the number of nodes, number of edges, number of cells, and number of edges for solid, viscous, and far-field boundaries. This is a simple, yet realistic, example of the data required to represent an unstructured grid.

Figure 9 depicts a program fragment which produces an ASCII file of type **unstruct_t**, and Figure 10 gives a analogous fragment for reading such a file. This example illustrates a few of the simplest SmartFile capabilities. First, fields can be read and written in different order, since the programmer is only concerned with the field abstractions, not their actual layout in the data file. Second, one of the fields is read repeatedly, while another is never read. There is complete freedom on reading; writing is more constrained. Every field specified by the file type must be written or the `sf_close` statement (in Figure 9) would fail on an incomplete type error.

```
main(){

sf_file *ugrid_file;

struct edge{
    int node1, node2, cell1, cell2;
} edges[EDGE_NO];

struct coord{
    double x, y;
} coords[NODE_NO];

err = sf_open("unstruct.dat", "Unstructured", SF_WRITE, SF_ASCII, &ugrid_file);
 ...
err = sf_bind(ugrid_file, "nnodes", nnodes);
err = sf_bind(ugrid_file, "nedges", nedges);
...
err = sf_put(ugrid_file,  "coords", coords, NODE_NO, NULL);
err = sf_put(ugrid_file,  "edgelist", edges, EDGE_NO, NULL);
 ...
err = sf_close(ugrid_file);
}
```

Figure 9: Program fragment writing unstructured grid data

```
main(){

sf_file *ugrid_file;
int nnodes;

struct edge{
    int node1, node2, cell1, cell2;
} edges[EDGE_NO];

struct coord{
    double x, y;
} coords[NODE_NO];

err = sf_open("ugrid_data", "Unstructured", SF_READ, SF_ASCII, &ugrid_file);
 ...
err = sf_get(ugrid_file,  "edgelist", edges, &nedges, NULL);
err = sf_get(ugrid_file,  "coords", coords, &nnodes, NULL);
...
err = sf_close(ugrid_file);
}
```

Figure 10: Program fragment reading unstructured grid data

9 Status and Conclusions

The goal of SmartFiles is to elevate the programmer to a higher level of abstraction, so that interaction with data files can be done at an abstract, conceptual level rather than at the level of `integers` and `floats`. In addition to the benefits of abstraction, it is also possible to do units conversion or other kinds of intelligent processing.

The SmartFile system is actively being developed at by a collaboration of computational scientists and computer scientists at the University of Wyoming and ICASE. Prototype software for the interface is available, and the DAFT compiler is currently being completed. We are also working on the integration of the udunits package that will allow for automatic units conversion based on the "units" attribute, and several unstructured CFD codes and their associated data files are begin converted to use the system. The support for legacy systems allows the old version of the programs to continue operating with the newer SmartFiles version.

Future plans for the system include developing additional conversion tables, interfacing with existing software systems for data files (such as netCDF), and automatic generation of layout descriptors for legacy systems. We are also planning to construct an MDO application for aircraft design using SmartFiles as the coupling devices between the codes. Finally, though performance has not been a major issue, we are looking at optimization techniques to improve the I/O system performance. For example, one idea is to flatten the fields with subfields so that the entire field (or multiple fields) can be read in a single operation.

In summary, we have designed and are currently implementing a system for applying object-oriented principles to data files. The goal of our work is to provide a solid, sensible approach to data file interoperability for scientific and engineering codes.

Acknowledgments

We wish to thank the anonymous reviewers for their insightful comments and Scott Leutenegger for his thoughts on object-oriented database systems.

References

[1] Malcom Atkinson, Francois Bancilhon, David DeWitt, Klaus Dittrich, David Maier, and Stanley Zdonik. The object-oriented database system manifesto. In *Proceedings of the 1st International Conference on Distributed and Object-Oriented Design*, pages 946–954, 1989.

[2] Ruth A. Aydt. The pablo self-defining data format. Technical report, Department of Computer Science, University of Illinois UC, March 1992. Revised July, 1994.

[3] Grady Booch. *Object-Oriented Analysis and Design.* The Benjamin/Cummings Series in Object-Oriented Software Engineering. Benjamin/Cummings, second edition, 1994. ISBN 0-8053-5340-2.

[4] Robin G. Fegeas, Janette L. Cascio, and Robert A. Lazar. An overview of fips 173, the spatial data transfer standard. *Cartography and Geographic Information Systems*, 19(5), December 1992. Special Issue: Implementing the Spatial Data Transfer Standard.

[5] John F. Karpovich, Andrew S. Grimshaw, and James C. French. Extensible file systems (elfs): An object-oriented approach to high performance file i/o. In *Proceedings of the 9th Annual Conference on Object-Oriented Programming Languages, Systems, and Applications*, pages 191–204, October 1994.

[6] NASA/Jet Propulsion Laboratory. *Planetary Data System Standards Reference Manual*, August 1994. Version 3.1.

[7] NASA/Science Office of Standards and Technology. *A User's Guide for the Flexible Image Transport System*, May 1994. Version 3.1.

[8] National Center for Supercomputing Applications (NCSA). *The HDF Reference Manual*, February 1994. Version 3.3.

[9] Russ Rew, Glen Davis, and Steve Emmerson. *NetCDF User's Guide*. Unidata Program Center, April 1993. Version 2.3.

[10] John D. Stackpole. *A Guide to GRIB: The World Meteorological Organization Form for the Storage of Weather Product Information and the Exchange of Weather Product Messages in Gridded Binary Form*. National Meteorological Center, National Weather Service, NOAA, 1.0 edition, February 1994.

[11] UNIDATA. Udunits: A library for manipulating units of physical quantity. http://www.unidata.ucar.edu/packages/udunits/.