deceit

January 12, 2017

Abstract

deceit is an interpreter to create arbitrary data sets complying with the DAL data model. Optionally, the dat set structures can be filled with data, either with the library functions provided with **deceit**, or with a user-defined library of Perl functions.

1 Instruments/Modes

Instrument	Mode
n/a	n/a

2 Use

pipeline processing	no
interactive analysis	no

3 Description

With **deceit** it is possible to create datasets (e.g., FITS files) without the need to do any complex programming. Within the constraints of the DAL data model, datasets of any complexity can be created. The datasets are described with a high level specification which may be thought of as a template. All the features of the DAL data model are supported. The template approach has the added advantage of documenting the dataset structure.

Throughout this document it is assumed that the reader has a basic understanding of the DAL data model.

By default, **deceit** creates the dataset empty; that is to say, the attributes are filled in as specified in the template, but none of the data structures are filled in with data.

Optionally, certain functions may be called from within the template. These functions allow one to use data contained in ASCII files to populate the arrays and columns of a dataset. It is possible for users to write their own Perl functions which may then be called from within a template. A number of Perl functions are provided that make use of the pedal, the Perl interface to the **dal**.

deceit cannot be used to modify an existing dataset.

3.1 A few examples

deceit is an interpreter, and its functionality can be completely described with a grammar. This is available in **??**. In the remainder of this section, the syntax of a **deceit** template is explained through a number of examples that cover most of the available functionality.

3.1.1 Getting started

This is the simplest **deceit** template:

```
# --> begin
# File: HelloWorld
#
# HelloWorld is the simplest case of a deceit template
dataset "HelloWorld.dat"
# --> end
```

Then running the command `deceit --file=HelloWorld` will generate the dataset `HelloWorld.dat`. It will contain no arrays, tables or attributes. The comment lines, which are skipped by the interpreter, are not required. This is essentially the simplest case of a dataset specification, and gives rise to the simplest DAL dataset.

Any number of dataset specifications may appear in a template.

3.1.2 A simple dataset

The following template describes a dataset with two dataset attributes and no blocks.

```
# --> begin
# a comment line. Blank lines, like the next one, are ignored.

dataset "simple.ds"
<
  attribute "ATT_STR" string "hello" "a familiar greeting" ""
  attribute "PI" real 3.1415 "a familiar number" "rad"
>
# --> end
```

The following syntax rules apply:

- Comment lines start with a `#` in the first column.
- White space (space, tabs, carriage return and new line characters) and comments are ignored by the interpreter.

- The first **deceit** command should be **dataset**, followed by the dataset name (a string).
- All strings are enclosed in double quotes.
- The dataset description is enclosed in $< \ldots >$.
- Attribute specifications have the following syntax:

  attribute *name type value label unit*

  where

  - *name* is the attribute name (a string).
  - *type* is the data type of the attribute. It may be one of **string real int bool**.
  - *value* is the attribute value. It may be one a quoted string, a real, an integer, or a boolean (1, true, 0, false).
  - *label* is a comment string.
  - *unit* is a unit string.

  All items are mandatory.

3.1.3 A dataset with an array (image) block

```
# --> begin
dataset "array.ds"
<
  attribute "ATT_STR" string "hello" "a familiar greeting" ""
  attribute "PI" real 3.1415 "a familiar number" "rad"

  array(10,10) "Small Image" real32 "a small image" "erg/cm**2/s"
  <
    attribute "INT" int 1999 "last chance before the end" "a"
  >
>
# --> end
```

The following syntax rules apply:

- Array specifications have the following syntax:

  array*(dim1, dim2, ...) name type label unit*

  where

  - *dim1, dim2* are integers demoting the dimensions of the array.
  - *name* is the array name (a string).
  - *type* is the type of the data stored in the array. It may be one of `int8 int16 int32 real32 real64`. FITS does not support boolean arrays.
  - *label* is a comment string.
  - *unit* is a unit string.

  All items are mandatory. If there are no array attributes, the $< >$ can be omitted.

3.1.4 A dataset with an table block of 10 rows and four columns

```
# --> begin
dataset "table.ds"
<
  attribute "ATT_STR" string "hello" "a familiar greeting" ""
  attribute "PI" real 3.1415 "a familiar number" "rad"

  table "TABLE" 10 "a small table"
  <
    attribute "GOOD" bool true "is the SAS good?" ""
    column "RAWX" int32 "raw CCD X coordinate" "pixel"
    column(32) "Strings" string "a string column" ""
    column(10,20) "MiniImages" real32 "images" "c/s"
    column(0) "VariableLength" int16 "as many as you line" "erg/s"
  >
>
# --> end
```

The following syntax rules apply:

- Table specifications have the following syntax:
  table *name rows label*
  where
  - *name* is the name of the table (a string).
  - *rows* is an integer specifying the number of rows in the table.
  - *label* is a comment string.
- Column specifications have the following syntax:
  column*(dim1, dim2, ...) name type label unit*
  where
  - *dim1, dim2, ...* are integers demoting the dimensions of the column. The dimension specifier can be omitted for normal columns. Variable length columns are given dimension 0. String columns can only have one dimension. If more than one dimension is specified, the column will be treated as a one-dimensional column with dimensions equal to the product of the of the dimensions given. This is a limitation of FITS.
  - *name* is the name of the column (a string).
  - *type* is the type of the data stored in the column. It may be one of `int8 int16 int32 real32 real64 string bool`.
  - *label* is a comment string.
  - *unit* is a unit string.

3.2 Detailed syntax

3.2.1 dataset

A **deceit** template comprises zero or more dataset specifications. The dataset specifications are placed into a file, and cannot appear in the body of any other specification. There is no limit to the number of dataset specifications which may appear in a template, but within a template each dataset name must be unique.

The general form of a dataset specification is as follows:

```
dataset dataset-name optional-expresion
<
         ... dataset item specifications ...
>
```

where,

- **dataset** is the dataset specification keyword. This keyword is used to begin a new dataset specification.
- *dataset-name* is the name of the dataset. There must be no other dataset specifications with the same name in the template. It has type *string-expression* (which may be a literal string). If the file exists it will be overwritten.
- *optional-expression* is an optional expression which is evaluated after the last item in the body has been evaluated.

The header part of the dataset specification (i.e., **dataset** *dataset-name optional-expression*) must be placed onto a single line. The subsequent dataset body, which is delimited by < and >, is optional but, if present, must begin on a new line. This is due to a limitation of the **deceit** interpreter.

The body of the dataset specification comprises zero or more of the following specifications:

- table
- array
- attribute
- history
- comment

Any number of these items may appear in a dataset specification.

The variable *dat* is a predefined variable, the value of which is a pointer to the dataset being created, and may be referenced in the optional expression, and also in the expressions of the dataset item specifications. For example,

```
dataset "set" verifyDataset(dat)
<
...
>
```

where **verifyDataset** might be a user-defined function (see 3.3) that takes a pointer to a dataset and, say, checks that the dataset conforms to some standard. The function is executed when the interpreter reaches the end of the dataset specification block.

3.2.2 Table

Table specifications may only appear in the body of a dataset specification. There is not limit to the number of table specifications which may appear in a dataset specification, but within a dataset table names should be unique.

The general form of a table specification is as follows:

```
table table-name rows label optional-expression
    <
        ... table item specifications ...
    >
```

where,

- `table` is the table specification keyword. This keyword is used to begin a new table specification.
- *table-name* is the name of the table. There must be no other table or array with the same name in the dataset. It has type *string-expression* (which may be a literal string).
- *rows* is the number of rows in the table. It has type *integer-expression* (which may be a literal integer).
- *label* is a short description. It has type *string*.
- *optional-expression* is an optional expression which is evaluated after the last item in the body has been evaluated.

The header part of the table specification (i.e., **table** *table-name optional-expression*) must be placed onto a single line. This is due to a limitation of the parser. The subsequent table body, which is delimited by < and >, is optional but, if present, must begin on a new line.

The table specification body comprises zero or more of the following specification:

- column
- attribute
- history
- comment

There is no limit to the number of items which may appear in a table specification.

The final optional expression can be used to direct **deceit** to execute a function. This function is executed when the interpreter reaches the end of the of the table specification block. The variable `tab` is a pointer to the table being created. For example,

```
dataset "set"
 <
 table "table" 100 "" fillTable(tab, "somefile")
```

```
        <
        ....
        >
    >
```

where **fillTable** is a user-defined (see 3.3) function that takes a pointer to a table, and fills it with data from `somefile`. The function is executed when the **deceit** interpreter reaches the end of the table specification block

3.2.3 Column

Column specifications may only appear in the body of a table specification. There is no limit to the number of column specifications which may appear in the table body. Within a table column names must be unique.

The general form of a column specification is given by

```
column column-dimensions column-name data-type label units optional-expression
 <
     ... column item specifications ...
 >
```

where,

- **column** is the column specification keyword. This keyword begins a new column specification.
- *column-name* is the name of the column. There must be no other column with the same name in the table. It has type *string-expression* (which may be a literal string).
- *column-dimensions*, denotes the dimensions of the column's cells.. It takes the form of a comma-separated list of integer-valued expressions, the list being delimited by parentheses. The general form is *(integer-expression, integer-expression, ...)*. For instance, (10,20) indicates that the columns cells are a two dimensional matrix with size 10 by 20. See 3.1.4 for string columns. Scalar, multi-dimensional and variable length columns are supported. Scalar and multi-dimensional columns have fixed length. Variable length columns are created by setting the dimensions to 0.
- *data-type* specifies the data type of the column. The available types are `int8, int16, int32, real32, real64, bool, string`.
- *label* is a short textual description for the column. It has type *string-expression* (which may be a literal string).
- *units* is string containing the column's units. Its type is *string-expression* (which may be a literal string). When data-type is bool or string the units must be the empty string. This appears to be a limitation of CFITSIO.
- *optional-expression* is an optional expression which is evaluated after the last item in the body has been evaluated.

The header part of the column specification (**column** *column-name optional-expression*) must be placed onto a single line. This is due to a limitation of the parser. The subsequent body, which is delimited by < and >, is optional but, if present, must begin on a new line.

The column specification body comprises zero or more of the following specifications:

- attribute

The final optional expression can be used to direct **deceit** to execut a function. The function is executed when the interpreter reaches the end of the column specification block. The variable *col* is a pointer to the column and may be referenced in the expression. For example,

```
dataset "set"
<
   table "table" 100
   <
       column "col" int32 "integer column" "m" fillColumn(col, "data.dat", 0)
    >
>
```

where **fillColumn** is a user-defined function (see 3.3) that takes a pointer to a column, and fills it with data from column 0 of `somefile`.

3.2.4 Array

Array specifications may only appear in the body of dataset specifications. There is no limit to the number of array specifications which may appear in the array specification body. Within a dataset, array names must be unique.

The general form of an array specification is given by

```
array array-dimensions name array-data-type label units optional-expression
<
    ... array item specifications ...
>
```

where,

- **array** is the array specification keyword. This keyword is used to begin a new array specification.
- *array-dimensions*, the dimensions of the array, is a comma-separated list of integer-valued expressions, the list being delimited by parentheses The general form is *(integer-expression, integer-expression, ...).*
- *name* is the name of the array. There must be no other table or array with the same name in the dataset. It has type *string-expression* (which may be a literal string).
- *array-data-type* specifies the data type of the array. The available types are `int8, int16, int32, real32, real64`.

- *label* is a short description for the array. It has type *string-expression* (which may be a literal string).
- *units* is the array's units. It has type *string-expression* (which may be literal string).
- *optional-expression* is an optional expression which is evaluated after the last item in the body has been evaluated.

The header part of the array specification **array** *array-dimensions name array-data-type label units optional-expression* must be placed onto a single line. This is due to a limitation of the parser. The subsequent body, which is delimited by < and >, is optional but, if present, must begin on a new line.

The array specification body comprises zero or more of the following specification:

- attribute
- history
- comment

The final optional expression can be used to direct **deceit** to execut a function. The function is executed when the interpreter reaches the end of the array specification block. The variable *arr* is a pointer to the array and may be referenced in the expression. For example,

```
dataset "ds1"
<
    array(10,20) "arr1" int32 "an integer array" "m" fillArray(arr,"somefile")
    <
       attribute "att1" int 10 "an integer attribute" "Nm"
    >
>
```

where **fillArray** is a user-defined function (see 3.3) that takes a pointer to an array, and fills it with data from `somefile`.

3.2.5 Attribute

Attribute specifications may appear in dataset, array, table and column body specifications. There is no limit to the number of specifications which may appear in each of the allowed specification bodies. Within each body section attribute names must be unique.

The general form of an attribute specification is given by

```
attribute attribute-name attribute-data-type value label units optional-expression
<
    ... attribute item specifications ...
>
```

where,

- **attribute** is the attribute specification keyword. This keyword is used to begin a new attribute specification.
- *attribute-name* is the attribute name. There must be no other attributes in the attribute's parent with the same name. It has type *string-expression* (which may be a literal string).
- *attribute-data-type* is the attribute data type. The available types are **int, real, bool, string**.
- *value* is the attribute value. It is an expression (which may be a literal value) which must evaluate to a value whose type is consistent with *attribute-data-type*.
- *label* is a short description. It has type *string-expression* (which may be a literal string).
- *units* is the attribute's units. It has type *string-expression* (which may be a literal-string).
- *optional-expression* is an optional expression which is evaluated after the last item in the body has been evaluated.

In general, the body of an attribute will not be needed.

3.2.6 History

History specifications may appear in the body of dataset, table and array specifications. There is no limit to the number of history specifications which may appear in each of the allowed specification bodies.

There are two allowed formats for the history specification. The first is the single string form whose general format is given by

```
history string
```

where

- **history** is the history specification keyword. This is used to begin a new history specification.
- *string* is the text which comprises the history. It must be a literal string.

The second form is given by

```
history < free-format-text >
```

where

- **history** is the history specification keyword. This is used to begin a new history specification.
- *free-format-text* is the text which comprises the history (delimited by < and >). It may be spread over several lines.

For example

```
dataset "ds1"
<
    history <
        Updated by a.n.other

        and a.u.thor
           >
>
```

3.2.7 Comment

Comment specifications may appear in the body of dataset, table and array specifications. There is no limit to the number of comment specifications which may appear in each of the allowed specification bodies.

There are two allowed formats for the comment specification. The first is the single string form whose general format is given by

```
comment string
```

where

- **comment** is the comment designator keyword. This is used to begin a new comment specification.
- *string* is the text which comprises the comment. It must be a literal string.

The second form is given by

```
comment < free-format-text >
```

where

- **comment** is the comment designator keyword. This is used to begin a new comment specification.
- *free-format-text* is the text which which comprises the comment (delimited by < and >). It may be spread over several lines.

For example:

```
dataset "ds1"
<
    comment "Updated on 31/12/99"
    comment <
       Updated on 1/12/99
    >
>
```

3.3 Perl functions provided with deceit

deceit is distributed with the following Perl functions that can be used to fill the dataset with data from text files.

- **rows**

  This function determines the number of rows in a text file. Blank lines, as well as line starting with # are not counted. The function takes one string argument, namely the name of the text file to be used.

  Example:

  ```
  table "Table" rows("table.dat") "a table"
  ```

- **fillColumn**

  The function signature is as follows: the first argument must be the variable `col`, the second the name of a text file from which the data is to be extracted, the third argument an integer denoting from which column in the text file data should be extracted. A column separator can be specified as an optional fourth argument. This is most useful when dealing with vector or string columns, as explained in the examples that follow. An optional fifth argument is a boolean (1/0) that indicates whether vector columns should be padded with zeroes if the text file does not contain enough elements to fill the row.

  Example of a scalar column:

  ```
  column "Column" int32 "integer column" "s" fillColumn(col, "column.dat", 0)
  ```

  Columns in the text file are delimited by white space. The first column is given number 0. Blank lines and lines starting with # are ignored.

  The file `column.dat` might look like this:

  ```
  1 10 100
  2 20 200
  3 30 300
  ```

  `fillColumn` will interpret this as a three-column ascii file. The data will be extracted from the first column (the first column is number 0).

  Example of a vector column:

  ```
  column(2) "Column" int32 "integer column" "s" fillColumn(col, "column.dat", 1, ":")
  ```

  The file `column.dat` might look like this:

  ```
  1 : 10 100 : 1000
  2 : 20 200 : 2000
  3 : 30 300 : 3000
  ```

  When the separator : is given, `fillColumn` will interpret this as a three-column ascii file. The second column (column number 1), from which the data are to be extracted, contains a vector of two elements. An error will result should any row in the second column contain more or less than 2 elements.

  At times it is acceptable to have less elements in the ascii file than cells in the dataset column. It is possible to instruct `fillColumn` to pad rows with trailing zeroes with a fifth parameter:

```
column(2) "Column" int32 "integer column" "s" fillColumn(col, "column.dat", 1, ":", 1)
```

Example of a two-dimensional vector column:

```
column(2,3) "Column" int32 "integer column" "s" fillColumn(col, "column.dat", 0, ":")
```

Vector columns in the text file are delimited by the separator character. Inside each vector column the individual elements are separated by white space. **deceit** follows the C convention for array indeces, i.e., the right-most index running faster. For instance, to fill a cell of dimensions (2,3) the elements must be stored as: (1,1) (1,2) (1,3) (2,1) (2,2) (2,3).

Example of a string column:

```
column(10) "Column" string "string column" "s" fillColumn(col, "column.dat", 0)
```

String columns are vector columns of characters. Multidimensional columns are not supported. That is, a string column with dimensions (2,3) is treated as one with dimensions (6). This is not a limitation of **deceit**. Strings in the text file should be quoted if they contain blanks. Strings are blank padded or truncated, as needed. A warning is issued in this case.

- **fillArray**

  This function takes a file name as its only argument. The file should contain the array data "unwrapped", with the left-most indices running faster (C convention).

- **fillConstColumn**

  This function takes two arguments: the variable `col` and a scalar. Each element in the column is set equal to the scalar specified.

- **fillConstArray**

  This function takes two arguments: the variable `arr` and a scalar. Each element in the array is set equal to the scalar specified.

4 Parameters

This section documents the parameters recognized by this task (if any).

Parameter	Mand	Type	Default	Constraints

file	yes	file	template.dct	

The text file describing the dataset.

withperllibrary	no	boolean	no	

Specifies whether the user wishes to supply a library of Perl functions.

perllibrary	no	file	deceit.pl	

The file containing the user-supplied library of Perl functions.

5 Errors

This section documents warnings and errors generated by this task (if any). Note that warnings and errors can also be generated in the SAS infrastructure libraries, in which case they would not be documented here. Refer to the index of all errors and warnings available in the HTML version of the SAS documentation.

Bad data type *(error)*
The template contains an unknown data type specifier.

Expected one of . . . *(error)*
The parser expected to find one of the items mentioned in the text of the message. This usually indicates a syntax error in the template.

6 Input Files

1. Text file containing the description of the dataset to be created.
2. Optionally, a library of user-defined Perl functions that should be used by **deceit**.

7 Output Files

1. Indirectly, **deceit** generates datasets that comply with the SAS data model.

8 Algorithm

The **deceit** parser is a top-down parser. Its behavior and algorithm is fully described by the **deceit** grammar.

9 Comments

- The user library functionality is not fully developed.
- The grammar is not specified.

10 Future developments

- When the datasets are filled using functions from the Perl library, **deceit** is quite slow. The performance bottleneck is in the pedal, the Perl interface to the DAL. This problem should be addressed in the pedal.

References