The File Structure Markup Language (FSML) Specification

Version 2.9

February 2004

Prepared by:
Science Applications International Corporation
Ship Technology Division
134 Holiday Court, Suite 318
Annapolis, Maryland 21401, USA

Contents

1 Introduction to FSML

This document presents the specification of the File Structure Markup Language or FSML. The sections below describe the motivation behind the development of FSML, as well as the concept for using FSML. Section 2 describes the structure of an FSML document and its constituent elements. Section 3 presents some examples of FSML documents. Finally, Section 4 lists the Document Type Definition (DTD) for FSML.

The current version is 2.9, released February 2004.

1.1 Purpose

Many strategies have emerged for integrating legacy applications into a current enterprise. Most of these strategies have focused on the development of "wrappers" (software that translates or maps between a common interface or interface description and the "wrapped" application). Occasionally the wrappers are written to make native calls to the applications. This is not always desirable or even feasible, however, as it involves modifying the legacy software and therefore requires a developer's knowledge that may have been lost. Because of this, the approach many wrappers take is to communicate with the legacy application via its input and output files. However, when input and output is complex – a typical situation for many applications, and engineering applications in particular – a substantial amount of knowledge and effort is required to make this wrapper function across the entire range of valid inputs and outputs. Often the scope of the interface is also reduced, resulting in lost functionality.

It is therefore desirable to have a way describe the structures of text files such that any valid file can be read and written without specialized software. Such a definition could be viewed as a computer-interpretable representation of knowledge about an application's interface. FSML was developed to satisfy this requirement. As a consequence, FSML can also be used for importing legacy application input and output files into a database or other data storage mechanism.

1.2 Concept

FSML takes advantage of the eXtensible Markup Language's hierarchical structure. This hierarchical structure gives XML and consequently FSML the ability to easily represent data structures as well as describe processes. FSML uses these capabilities to capture file parameters, data types, and semantics, as well as the process to be used for reading and writing data to the file.

Figure 1 shows how the information captured in an FSML document might be used. In the figure, a legacy application generates output files (A and B) whose structures have been described by the corresponding FSML documents. An FSML Processor (software that interprets FSML to read and write file structures described by FSML) can then read the files generated by the application and stores that information in memory. The processor can next be directed to write input files (C and D) for another application based on the corresponding FSML documents using data from the read output files, and data supplied by other means (user, database, etc.). Another legacy application can then run using those generated input files.

Any number of applications – legacy or current – could communicate in this way. In addition, the FSML Processor could be linked to databases and other non-legacy applications that use the data extracted from the output files of a legacy application, and/or supply input for another legacy application.

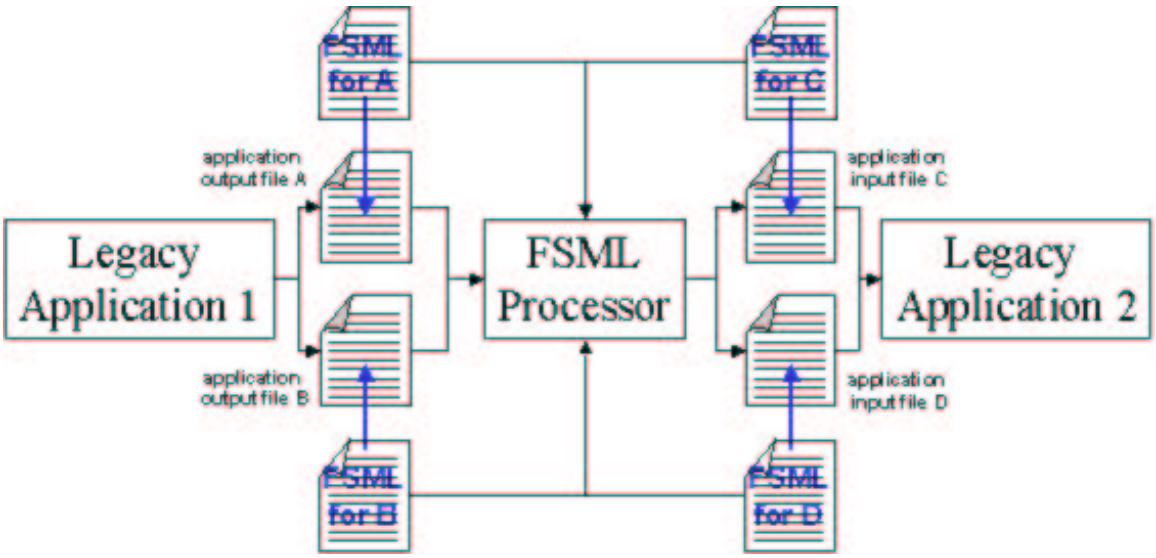

Figure 1: FSML Concept

It is vital that the user understand FSML's basic progression. The processor reads a file, stores that data in memory, (optionally) manipulates that data, and then (optionally) writes another file. That means that all data to be written must either reside in memory (after being read from the first file) or be hardcoded in the FSML file. Always remember that FSML simultaneously describes how to read and how to write.

1.3 Documentation Conventions and Terminology

XML Structure and Syntax
FSML uses XML structure and syntax, so the most common XML terms used in this document are defined below for the reader's convenience. For more information about authoring XML documents, see the XML specification at http://www.w3c.org/TR/REC-xml.

Element: A declaration of an item from the document's Document Type Definition (DTD). Also called a tag. The syntax of an element is as follows:

For elements that do not contain other elements –

```
<ElementName Attributes/>
```

For elements that contain other elements –

```
<ElementName Attributes> OtherElements </ElementName>
```

An example is given below, under "Examples."

Attribute: A setting or configuration of an element. The syntax of an attribute is as follows:

```
Attribute="value"
```

Tables
This document uses tables to define the attributes that should be specified for each element. The "Required" column indicates whether or not an attribute must be specified.

Attribute Name	Description	Required
type	Name of the file type	no
description	Description of the file type	no
etc.		

Examples
This document also includes examples of how these elements look in an actual document. These examples are depicted as follows:

Example

```
<FileTypeSpecification type="LAMP Control File" lineCommentStart="!" />
```

This declares a FileTypeSpecification named "LAMP Control File". LAMP Control Files can include line comments starting with "!", but no block comments

2 Document Structure

An FSML document consists of three major components:

- The file begins with several lines of declarations and a root element, which describe the XML version, document type, file type, and basic formatting information about the file to be read.
- The FileData section defines all the parameters in the file including data type, location within the hierarchical structure of parameters, and documentation.
- The FileStructure section describes the process to be used for reading and writing the file data.

These components fit together in the following basic structure:

Structure	**Sample FSML Code**
Declarations	`<?xml version="1.0" encoding="UTF-8"?>`
	`<!DOCTYPE FileTypeSpecification SYSTEM "file:`*`filename`*`">`
FileTypeSpecification begins	`<FileTypeSpecification` *`Attributes`*`>`
FileData begins	`<FileData>`
	`Elements`
FileData ends	`</FileData>`
FileStructure begins	`<FileStructure>`
	`Elements`
FileStructure ends	`</FileStructure>`
FileTypeSpecification ends	`</FileTypeSpecification>`

The next sections discuss each of these FSML requirements in depth:

- declarations and FileTypeSpecification are described in Section 2.1
- file data are described in Section 2.2
- file structure is described in Section 2.3.

In addition, some example FSML documents are provided in Section 3.

2.1 Declarations and FileTypeSpecification

All valid FSML documents start with the XML Declaration and the location of the Document Type Declaration (DTD). Below is an example of the declarations lines:

```
<?xml version="1.0" ?>
<!DOCTYPE FileTypeSpecification SYSTEM "file:///d:\users\wood\DASI\FSMLv2.9.dtd">
```

Section 4 shows the DTD for FSML, quoted in the example above as a file named `d:\users\wood\DASI\FSMLv2.9.dtd`.

Next, the root element, FileTypeSpecification, should be declared. The FileTypeSpecifica-

tion has the following attributes:

Attribute Name	Description	Required
type	Name of the file type	no
description	Description of the file type	no
lineCommentStart	Character that indicates the entire line which follows is a comment in the file	no
blockCommentStart	Character that indicates the beginning of a block comment in the file *	no
blockCommentEnd	Character that indicates the end of a block comment in the file *	no
ignoreBlankLines	True/False. "True" strips out blank lines in preprocessing; if blanks are meaningful, set to "False"	no
delimiters	Characters that indicate a new piece of data; default is whitespace **	no
ignoreMissingValues	True/False. If a missing value is encountered, "True" will log an error and continue processing; "False" causes the process to fail	no

Example

```
<FileTypeSpecification type="LAMP Control File" lineCommentStart="!" />
```

This declares a FileTypeSpecification named "LAMP Control File". LAMP Control Files can include line comments starting with "!", but no block comments

---

The FileTypeSpecification element must contain the FileData and FileStructure elements in turn. These two elements are detailed in Sections 2.2 and 2.3.

The FSML file must end with a tag to close the FileTypeSpecification, in the format `</FileTypeSpecification>`.

* A note about comments: Comments in the original application file are stripped as part of FSML preprocessing, using the blockCommentStart and blockCommentEnd attributes. The stripped comments are not retained. Therefore, comments will not be preserved in a file written by the FSML processor.

** A note about delimiters: FSML automatically uses whitespace as the delimiter when reading and writing files. Examples of whitespace are tabs, line breaks, and any number of blank spaces (" "). The default is whitespace. If your file is delimited by a character other than whitespace, then the `delimiters` attribute must be set; for example, a comma-delimited file would require an attribute of `delimiters=","`. Note that the specified delimiter is *in addition to* the default whitespace. Use of the `delimiters` attribute is rare.

2.2 The FileData Section

2.2.1 Overview of the <FileData> Tag

The FileData section defines all the parameters in the legacy file; the process used for reading and writing these parameters will then be defined in the FileStructure section (Section 2.3).

The FileData section is declared using the FileData element. The FileData element has no attributes.

Example

```
<FileData>
   <String name="PRNAME"/>
   <Float name="AGE" units="days"/>
</FileData>
```

This FileData structure declares two data items: a string named "PRNAME" and a floating number named "AGE"

The FileData section must contain one or more String, Integer, Float, or Array elements in any order. The following attributes are common to each of these elements:

Attribute Name	Description	Required
name	Name of the parameter. This name is used as an identifier so other parts of the document can reference this parameter; therefore, it must be unique	yes
description	Description and/or documentation of this parameter. Note that this will be treated as markup, which means that HTML entities will have to be used to represent reserved characters such as ¡, ¿, etc.	no
units	Units of this parameter	no
group	Defines which parameters group this parameter belongs to. Might be used by the processor to display parameters hierarchically	no

2.2.2 String

A String element declares a String parameter in the file. A String can be any combination of letters, numbers, and special characters (except for blank spaces, XML-reserved characters, and characters defined as delimiters in the FileTypeSpecification root element). Note that – at the present time – if the target string contains a blank space, underline, or any other special character (for example, `'Container Ship in sea state 5'`, then it cannot be read as one string; it must be read as a series of strings (one for `Container`, one for `Ship`, etc.) or as formatted data (as `A9,A1,A4,A1,A2,A1,A3,A5,A1,A1`; for more information on processing formatted data, see Section 3.4).

The String element has no additional attributes beyond the basics listed in Section 2.2.1.

Example

```
<String name="FAPLT" description="File name for autopilot data.  If a
file is specified, then the autopilot data is read from it in the format
described in the User's Guide.  If this entry is empty, defaults for
the autopilot variables will be used.  The file name may be up to 40
characters long and can include an absolute or relative path name."
group="Input Files" />
```

This declares a String parameter called "FAPLT" belonging to the group "Input Files"

A more in-depth example is given in Section 3.1.1.

2.2.3 Integer

An Integer element declares an Integer parameter in the file. The Integer element has no additional attributes beyond the basics listed in Section 2.2.1.

Example

```
<Integer name="NXFS" description="Number of panels to generate on
the free surface; enter 0 to use matching surface panel distribution"
group="Free Surface Settings" />
```

This declares an Integer parameter called "NXFS" belonging to the group "Free Surface Settings"

A more in-depth example is given in Section 3.1.2.

2.2.4 Float

A Float element declares a Float parameter in the file. Note that a Float element can be used to read floating point numbers *and* integers. The Float element has no additional attributes beyond the basics listed in Section 2.2.1.

Example

```
<Float name="DTH" description="Time step interval for calc (&gt;0.0)."
group="Time Stepping Settings" />
```

This declares a Float parameter called "DTH" belonging to the group "Time Stepping Settings" with a description of "Time step interval for calc (>0.0)." Note the use of the ">" entity, which is markup for the ">" symbol; this is required since the description is treated as markup and ">" is reserved for the end of an element.

A more in-depth example is given in Section 3.1.3.

2.2.5 Array

The Array element declares a parameter array in the file.

An array can be processed in three different ways. Refer to the following snippet of data:

```
57.43  -183.67  0.0  300.00  mild
57.42  -182.96  0.0  301.00  mild
57.41  -182.26  0.0  302.00  mild
57.40  -181.55  0.0  303.00  mild
57.39  -180.85  0.0  304.00  mild
57.38  -180.14  0.0  305.00  moderate
57.37  -179.44  0.0  306.00  moderate
```

The three ways to process this data are:

- using one-dimension arrays in the vertical direction, so that there are five arrays: 57.43 to 57.37, -183.67 to -179.44, 0.0 to 0.0, 300.00 to 306.00, and mild to moderate.
- using one-dimension arrays in the horizontal direction, so that there are seven arrays: 57.43 to mild, 57.42 to mild, 57.41 to mild, and so on, ending with 57.37 to moderate.
- using a two-dimensional array and reading both directions at once.

The processing method you choose will depend on the data.

The Array element has the following additional attributes:

Attribute Name	Description	Required
ndims	The number of dimensions in this array. Note that this can be a parameter name, in which case the value of that parameter will be used	yes
dims	A comma-delimited list of the maximum index for each dimension. Note that an item in the list can be a parameter name, in which case the value of that parameter will be used	yes
baseType	The data type for all elements in this array. Must be String, Integer, or Float	yes

Example

```
<Array name="AIS" description="Y-Z, Y, and Z mass moments of inertia
of the weight station about the points specified by XMS. Dimensions of
mass moment of inertia (rho*L^5). Positive value required." ndims="2"
dims="3,NBMX" group="Load Calculation Settings" baseType="Float" />
```

This declares a 2-dimensional array-of-floats parameter named “AIS”. The first dimension is of size 3, and the second dimension has size equal to the value of the NBMX parameter. The parameter belongs to the Load Calculation Settings group

In-depth examples of processing arrays, with portions of input files and corresponding FSML coding, are given in Section 3.2.1 (vertical arrays), Section 3.2.2 (horizontal arrays), and Section 3.2.3 (multi-dimensional arrays).

2.3 The FileStructure Section

2.3.1 Overview of the <FileStructure> Tag

The FileStructure section defines the process used for reading and writing the file, including the parameters defined in the FileData section (Section 2.2).

This section is declared using the FileStructure element. The FileStructure element has no attributes.

Example

```
<FileStructure>
   <StaticText>!  Data file for program evaluator</StaticText>
   <Record ignoreOnRead="false">
      <SetValue target="PRNAME"/>
   </Record>
   <Record ignoreOnRead="false">
      <SetValue target="AGE"/>
   </Record>
   <StaticText>!  * * * End of File * * *</StaticText>
</FileStructure>
```

This FileStructure writes a line of text, reads two records, and writes another line of text

---

The FileStructure section must contain one or more structure elements in any order. The structure elements are:

- Record
- Condition
- Loop
- StaticText.

Each of these structure elements must contain one or more operation elements in any order. The operation elements are:

- Separator
- SetValue
- AddValue
- SetDims
- SetSize.

Structure elements are discussed in Sections 2.3.2 through 2.3.5, and operation elements are discussed in Sections 2.3.6 through 2.3.10.

2.3.2 Structure Element: Record

The Record element defines a line of data in a file. Record has the following attributes:

Attribute Name	Description	Required
ignoreOnRead	True/False. "False" indicates that moving to the next line is required before performing the operations contained in the Record. When writing, a new line will always be created before performing operations	yes
format	The format statement that should be used on the record. Can be a parameter that specifies the format statement or the format statement itself	no
ignoreFormatOnRead	True/False. "True" ignores the specified format and reads values as encountered	no

A Record element must contain one or more structure elements such as Condition or Loop, or operation elements such as Separator, SetValue, SetSize, SetDims, or AddValue in any order.

Example 1

```
<Record ignoreOnRead="false">
   <SetValue target="DESCR" />
</Record>
```

Indicates that moving to a new line is required before setting the value of DESCR

Example 2

```
<Record format="(2F6.2,A2)" ignoreOnRead="false" ignoreFormatOnRead="true">
   <SetValue target="XCOORD" />
   <SetValue target="YCOORD" />
   <SetValue target="ID" />
</Record>
```

Reads three values in fixed format: a float called "XCOORD", a float called "YCOORD", and a two-character sequence called "ID"

Example 3

Source Code Fragment

```
   line 16:  (A)
   line 17:  Run2
```

```
<Record ignoreOnRead="false" >
   <SetValue target="RunTitleFormat" />
</Record>
<Record ignoreOnRead="false" format="RunTitleFormat" >
   <SetValue target="RunTitle" />
</Record>
```

Reads the value of "RunTitleFormat" (example: `(A)`), then applies that format when reading the value of "RunTitle" (example: `Untitled`)

2.3.3 Structure Element: Condition

A Condition element indicates that the structure and/or operations contained within the Condition element should be processed only if the condition evaluates to true. The condition inside a Condition element must be a simple condition; however, Condition elements can be nested to achieve complex conditions joined by AND. For complex conditions joined by OR, parallel condition elements can be used. The processor will process any structure or operation defined in a satisfied condition.

The Condition element has the following attributes:

Attribute Name	Description	Required
check	The name of the parameter that is being checked against the condition. This parameter's value must have been set previously	yes
operator	The relational operator to be used. Must be GT (greater than), LT (less than), or EQ (equal to)	yes
checkValue	The value to check against. This can be the name of a parameter whose value has been set previously	yes

Example

```
<Condition check="ISEA" operator="EQ" checkValue="1">
   <SetValue target="NWAVES" />
   <SetValue target="NWSC" />
</Condition>
```

Indicates that if the value of ISEA (which has been previously read) equals 1, then the two SetValue operations should be performed

A more in-depth example is given in Section 3.6.

2.3.4 Structure Element: Loop

The Loop element indicates that the contained structure and/or operations should be repeated a certain number of times. Loop elements can be nested in order to populate a multi-dimensional array. Loops can be set up in four different ways, which can be combined as needed:

- fixed length
- read the number of elements that was previously specified in the file (see Example 1 below, with NWAVES)
- read until a stop value is encountered (see Example 2 below)
- read until an end-of-file is encountered (see Example 2 below).

The Loop element has the following attributes:

Attribute Name	**Description**	**Required**
loopVariable	Name of a local variable that stores the value of the loop index (iteration)	no
start	The starting value for the loop index. This can be an integer or the name of a parameter whose value had been set previously	yes
end	The ending value for the loop index. This can be an integer or the name of a parameter whose value had been set previously	yes
incr	The increment value for the loop index. This can be an integer or the name of a parameter whose value had been set previously	yes
check	Value to be checked as part of the stop condition	no
operator	The relational operator to be used. Must be GT (greater than), LT (less than), or EQ (equal to)	no
checkValue	The value to check against. This can be the name of a parameter whose value has been set previously	no
stopOnEof	True/False. If an end-of-file is encountered, a value of "True" will throw an exception but the method returns normally; set to "True" if an EOF is expected	no
doOnce	True/False. "True" means that, even if the checkValue fails, execute the loop once	no

The Loop element must contain one or more structure elements such as Condition or another Loop, or one or more operation elements such as Separator, SetValue, SetSize, SetDims, or AddValue in any order.

Example 1

```
<Loop start="1" end="NWSC" incr="1">
   <Record ignoreOnRead="true">
      <AddValue target="WSCSTP" />
      <AddValue target="WSCFAC" />
   </Record>
</Loop>
```

Indicates that a number of records equal to the value of the NWSC parameter should be read. Note the value of ignoreOnRead, which means that a new line is not required before reading the values of WSCSTP and WSCFAC

Example 2

```
<Loop loopVariable="i" start="0" end="200" incr="1" check="SEQ[i]"
operator="NE" checkValue="-9" stopOnEof="true" doOnce="true" >
   <Record ignoreOnRead="false" format="EventFormat"
ignoreFormatOnRead="false">
      <AddValue target="SEQ" />
      <Condition check="SEQ[i]" operator="NE" checkValue="-9">
         <AddValue target="TSTRT" />
         <AddValue target="EVNT_NAME" />
         <AddValue target="IDX1" />
               .  .  .
         <AddValue target="MIN" />
         <AddValue target="MAX" />
         <AddValue target="Comments" />
      </Condition>
   </Record>
</Loop>
```

This loops to read an array (previously named "SEQ") of unknown size between 0 and 200, ended by the value "–9". Before looping, the Loop element checks that the loopVariable counter does not equal 200, the current Record value does not equal –9, and an EOF has not been encountered. If any of these operator conditions is met, the Loop ends

Another example, with a portion of an input file, is given in Section 3.3.

2.3.5 Structure Element: StaticText

The StaticText element defines a line of text that should always be written to the file, such as a comment. The StaticText element has no attributes and may contain the line of text to be written to the file.

Example

```
<StaticText>!01 DESCR - Descriptive Title (max 80 char)</StaticText>
<Record ignoreOnRead="false">
   <SetValue target="DESCR" />
</Record>
```

Indicates that the text "!01 DESCR - Descriptive Title (max 80 char)" should be written to the file before the record containing the DESCR parameter

A more in-depth example is given in Section 3.5.

2.3.6 Operation Element: Separator

The Separator element is used primarily as a formatting device to insert some characters, such as a blank or a comma, within a record. Note that FSML automatically writes one blank space (" ") between each element in a record, so the Separator element is necessary only if you want a different separator (like a dash) or more than that one blank space.

The Separator element has the following attributes:

Attribute Name	Description	Required
ignoreOnRead	True/False. Indicates whether this separator should be should or should not be accounted for when reading the file. It will always be written to the file	yes
string	Indicates the character or characters that should be used in this separator. A zero-length string means that a space should be used	yes
repeat	Indicates the number of times the value of string should be repeated in this separator	no

The Separator element contains no other elements.

Example

```
<Separator ignoreOnRead="true" string="" repeat="4" />
```

Indicates that four spaces should be written to the file, although they may not necessarily be present when reading

2.3.7 Operation Element: SetValue

The SetValue element indicates that the value of a parameter should be read from or written to the file. The SetValue element has the following attribute:

Attribute Name	Description	Required
target	Name of the parameter whose value should be read or written	yes

The SetValue element contains no other elements.

Example

```
<SetValue target="NXFS" />
```

Indicates the value of parameter "NXFS" should be read from or written to the file

2.3.8 Operation Element: AddValue

When reading a file, the AddValue element indicates that the data read should be added to an array. When writing a file, the AddValue element indicates that the software should loop through the array, retrieve the next value, and write that value to a file.

To create a general representation of arrays with arbitrary numbers of dimensions and dimensional sizes, it is necessary that all arrays be mapped to an equivalent one-dimensional array while reading and writing. Consecutive calls to AddValue will access items in the array by iterating over the end dimension first, the end–1 dimension second, and so on. For a 3x4 array, consecutive calls to AddValue while reading would set the array values in the following order: a11,a12,a13,a14,a21,a22,a23,a24,a31,a32,a33,a34.

The AddValue element has the following attribute:

Attribute Name	Description	Required
target	The name of the array to which this value should be added or from which should be written	yes

The AddValue element contains no other elements.

Example

```
<AddValue target="AUXOUT" />
```

Indicates that when reading, the data should be added to the array named "AUXOUT"

2.3.9 Operation Element: SetDims

The SetDims element indicates that an array's dimensions should be set. This is used when the dimensions of an array may change, depending on the values of other parameters in the file. SetDims sets the array's dimensions all at once; this is different from SetSize (see Section 2.3.10), which sets the dimensions one at a time. The SetDims element has the following attributes:

Attribute Name	Description	Required
target	The name of the array for which to set the dimensions	yes
dims	A comma-delimited list of the maximum index for each dimension. Note that an item in the list can be an integer or a parameter name provided that the value of that parameter has been set, in which case the value of that parameter will be used	yes

The SetDims element contains no other elements.

Example

```
<SetDims target="FREQW" dims="10,10,DEPTH" />
```

Indicates that the array named "FREQW" is a three-dimensional array with maximum indices of 10, 10, and the value of the parameter "DEPTH"

2.3.10 Operation Element: SetSize

The SetSize element indicates that one of an array's dimensions should be set. This is used when a dimension may change, depending on the values of other parameters in the file. SetSize sets the dimensions one at a time, whereas SetDims sets the dimensions all at once (see Section 2.3.9). The SetSize element has the following attributes:

Attribute Name	Description	Required
target	The name of the array for which to set the size	yes
dimIndex	The index of the dimension for which the size is being set. This can be an integer or a parameter name provided that the value of that parameter has been set, in which case the value of that parameter will be used	yes
size	The size of the dimension (max index +1). This can be an integer or a parameter name provided that the value of that parameter has been set, in which case the value of that parameter will be used	yes

The SetSize element contains no other elements.

Example

```
<SetSize target="FREQW" dimIndex="1" size="6" />
```

Indicates that the size of dimension 1 in the parameter array named "FREQW" should be 6

3 Examples

3.1 How to Process a Scalar Value

For the purpose of FSML coding, scalar values are considered to be strings, integers, and floating point numbers. The examples in the next sections discuss how to code FSML for each of these data types, with snippets from input files and FSML files.

3.1.1 How to Process a String

A string is a linear alpha-numeric phrase. A string must be defined in the FileData section and then read by the FileStructure section. This example processes three records, each of which is treated as a string.

Snippet from the Input Deck

```
Broadreach
Kennedy
2003
```

Snippet from the FSML File

```
<FileData>
  <String name="FerryLine"/>
  <String name="FerryName"/>
  <String name="CommDate" description="Date of commissioning"/>
</FileData>

<FileStructure>
  <Record ignoreOnRead="false"/>
    <SetValue target="FerryLine"/>
  </Record>
  <Record ignoreOnRead="false"/>
    <SetValue target="FerryName"/>
  </Record>
  <Record ignoreOnRead="false"/>
    <SetValue target="CommDate"/>
  </Record>
</FileStructure>
```

3.1.2 How to Process an Integer

As explained in Section 2.2.3, an integer parameter is declared by the Integer element. Below is an example that reads six integers from two different lines (records).

Note that the order in the FileData section is completely independent of the order in the FileStructure section; just be sure that every element to be read in the FileStructure section is defined in the FileData section.

Snippet from the Input Deck

```
322   68   2100
320   68   2100
```

Snippet from the FSML File

```
<FileData>
  <Integer name="lenD"  description="length, feet, designed"
                        group="Design"/>
  <Integer name="lenB"  description="length, feet, as built"
                        group="As Built"/>
  <Integer name="beamD" description="beam, feet, designed"
                        group="Design"/>
  <Integer name="beamB" description="beam, feet, as built"
                        group="As Built"/>
  <Integer name="dispD" description="displacement, tons, designed"
                        group="Design"/>
  <Integer name="dispB" description="displacement, tons, as built"
                        group="As Built"/>
</FileData>

<FileStructure>
  <Record ignoreOnRead="false"/>
    <SetValue target="lenD"/>
    <SetValue target="beamD"/>
    <SetValue target="dispD"/>
  </Record>
  <Record ignoreOnRead="false"/>
    <SetValue target="lenB"/>
    <SetValue target="beamB"/>
    <SetValue target="dispB"/>
  </Record>
</FileStructure>
```

3.1.3 How to Process a Floating Point Number

Non-integer numbers are declared with the Float element. Note that Float elements can also read an integer, but Integer elements cannot read a floating point number.

Snippet from the Input Deck

```
17.4
80
72.5
```

Snippet from the FSML File

```
<FileData>
  <Float name="dist" description="crossing distance" units="miles"/>
  <Float name="avgtime" description="average crossing time" units="minutes"/>
  <Float name="mintime" description="fastest crossing time" units="minutes"/>
</FileData>

<FileStructure>
  <Record ignoreOnRead="false"/>
    <SetValue target="dist"/>
  </Record>
  <Record ignoreOnRead="false"/>
    <SetValue target="avgtime"/>
  </Record>
  <Record ignoreOnRead="false"/>
    <SetValue target="mintime"/>
  </Record>
</FileStructure>
```

3.2 How to Process Array Values

As described in Section 2.2.5, all arrays are defined in the FileData section and then processed by loop(s) in the FileStructure section.

The next sections give examples of the three ways to read an array: vertically in Section 3.2.1, horizontally in Section 3.2.2, or multi-dimensionally in Section 3.2.3.

3.2.1 Vertical Arrays (m x 1)

A vertical array is simply an array that is 1 value wide and M values deep. M can be defined in the FSML code or can be read from the input file. In the simple example below, the value of M is read from the file and that value, defined as `numships`, is used to count through a loop three times.

For a more complex example of vertical arrays, see the loop example in Section 3.3. In the loop example, the data is arranged in a multi-column data file. The data is processed as three vertical arrays, each of which is 101 records long.

Snippet from the Input Deck

```
Presidential
3
Kennedy
Monroe
Harrison
```

Snippet from the FSML File

```
<FileData>
  <String name="shipclass"/>
  <Integer name="numships"/>
  <Array name="shipname" ndims="1" dim="numships" baseType="String"/>
</FileData>

<FileStructure>
  <Record ignoreOnRead="false"/>
    <SetValue target="shipclass"/>
  </Record>
  <Record ignoreOnRead="false">
    <SetValue target="numships"/>
  </Record>
```

```
  <Loop loopVariable="i" start="1" end="numships" incr="1" doOnce="false">
    <Record ignoreOnRead="false"/>
      <AddValue target="shipname"/>
    </Record>
  </Loop>
</FileStructure>
```

3.2.2 Horizontal Arrays (1 x n)

A horizontal array is simply an array that is 1 value deep and N values wide. N can be defined in the FSML code or can be read from the input file. In the example below, a loop reads the six values of the `engnames` array and then another loop reads the six values of the `engyears` array. The second loop also includes a Separator element so that the integer values for `engyears` are widely spaced for easier viewing (this could also be accomplished by one String element that was four blank spaces long).

Snippet from the Input Deck

```
! Engineers
Koundouriotis Li MacDonald Passche Smith Takata
1             3  2         16      4     2
```

Snippet from the FSML File

```
<FileData>
  <Array name="engnames" description="Names of Engineering Staff"
         ndims="1" dim="6" baseType="String"/>
  <Array name="engyears" description="Years of Service"
         ndims="1" dim="6" baseType="Integer"/>
</FileData>

<FileStructure>
  <StaticText>! Engineers</StaticText>
  <Record ignoreOnRead="false"/>
    <Loop start="1" end="6" incr="1">
      <AddValue target="engnames"/>
    </Loop>
  </Record>
  <Record ignoreOnRead="false"/>
    <Loop start="1" end="6" incr="1">
      <AddValue target="engyears"/>
    <Separator ignoreOnRead="true" string=" " repeat="4"/>
    </Loop>
  </Record>
</FileStructure>
```

3.2.3 Multi-Dimensional Arrays (m x n)

This example shows how an array of data can be read vertically and horizontally at the same time, by nested loops. The outer loop reads the vertical RPM values, up to the maximum set by the NUMRPM entry. The inner loop reads the horizontal speed values, up to the maximum set by the NUMSPEED entry. In both cases, the Separator elements are used to space the values for easier reading by people.

Snippet from the Input Deck

```
NUMRPM   5
NUMSPEED 3
RPM    NEAPSPD  INSPEED  OUTSPEED
1000   13.3     13.9     8.8
1500   15.0     15.6     10.2
2000   16.9     17.4     12.0
2500   17.5     19.9     15.1
3000   19.0     22.4     17.4
```

Snippet from the FSML File

```
<FileData>
  <Integer name="numrpm" description="number of RPM entries in table"/>
  <Integer name="numspeed" description="number of speed columns in table"/>
  <Array name="rpm" ndims="1" dims="numrpm" baseType="Integer"
         description="engine revolutions per minute"/>
  <Array name="speed" ndims="1" dims="numspeed" baseType="Float"
         description="speed at this tide condition" units="knots"/>
</FileData>

<FileStructure>
  <Record ignoreOnRead="false">
    <StaticText>NUMRPM   </StaticText>
    <SetValue target="numrpm"/>
  </Record>
  <Record ignoreOnRead="false">
    <StaticText>NUMSPEED </StaticText>
    <SetValue target="numspeed"/>
  </Record>
  <Record ignoreOnRead="false">
    <StaticText>RPM    NEAPSPD  INSPEED  OUTSPEED</StaticText>
  </Record>
  <Record ignoreOnRead="false">
    <Loop loopVariable="i" start="1" end="numrpm" incr="1">
```

```
      <AddValue target="rpm"/>
      <Separator ignoreOnRead="true" string="" repeat="3"/>
      <Loop loopVariable="j" start="1" end="numspeed" incr="1">
        <AddValue target="speed"/>
        <Separator ignoreOnRead="true" string="" repeat="5"/>
      </Loop>
    </Loop>
  </Record>
</FileStructure>
```

3.3 How to Process a Loop, and Looping Controls

As explained in Section 2.3.4, a loop repeats a certain operation under controlled conditions. In this example, the loop reads a column-oriented data file as three vertical arrays: one for time, one for temperature, and one for humidity. The number of records is the first entry in the input file. The FSML reads the first entry (in this case, 101) and designates it as the endpoint for the looping control. The loop then checks whether that endpoint has been reached; if not, it reads the next line of three arrays, increments the counter by 1 (set with the attribute for `incr`), and repeats.

Snippet from the Input Deck

```
101
00000  283.0000  76.2154
00010  283.0100  75.9284
00020  283.9514  74.5295
00030  284.6030  71.0012
00040  286.5713  68.6293
...
00980  313.5561  45.2142
00990  315.1357  44.2348
01000  317.9414  43.8304
```

Snippet from the FSML File

```
<FileData>
  <Integer name="numpoints"/>
  <Array name="time" ndims="1" dims="numpoints" baseType="Integer"/>
  <Array name="tempK" ndims="1" dims="numpoints" baseType="Float"/>
  <Array name="humid" ndims="1" dims="numpoints" baseType="Float"/>
</FileData>

<FileStructure>
  <Record ignoreOnRead="false">
    <SetValue target="numpoints"/>
  </Record>
  <Loop loopVariable="i" start="1" end="numpoints" incr="1" doOnce="false">
    <Record ignoreOnRead="false">
      <AddValue target="seconds"/>
      <AddValue target="tempK"/>
      <AddValue target="humid"/>
    </Record>
  </Loop>
</FileStructure>
```

3.4 How to Process Formatted Data

Formatted data involves multiple kinds of data together. In this example, we will read latitude and longitude data for a target. Each direction is expressed in degrees, minutes, and hemisphere, so that `44-20.00-N` means a latitude of 44 degrees and 20.00 minutes in the Northern hemisphere.

In the FileData section, strings are designated for the variable names `targlat` and `targlong`. An integer is then designated for each of the data parts in the formatted latitude/longitude data: latitude degrees, latitude minutes, latitude hemisphere, longitude degrees, longitude minutes, and longitude hemisphere.

In the FileStructure section, one record is programmed to read the latitude and one record is programmed to read the longitude. For the latitude, the entry in the input deck is `targlat  44-20.00-N`: in other words, a seven-letter string, then four blank spaces, then a two-digit integer, then a dash acting as a delimiter, then a floating number in the format of two-digits/decimal/two-digits, then a dash acting as a delimiter, and finally a one-letter string. This translates into a format of (A6,4X,I2,A1,F5.2,A1,A1). Once the format is set, then each of the latitude components can be read with a separate SetValue. This process is repeated for longitude (note that the format changes slightly, since `targlong` is an eight-letter string).

(A note about reading formatted data that contains delimiters: if the FSML file will be both reading *and* writing, then the processor must specifically read the delimiter into memory. This is the case with the example here, where the dash is a delimiter and is defined as format A1. Defining the delimiter as A1 stores the dash in memory so it can be written to a file. If, however, the FSML file is only reading, then the delimiter could be skipped: it does not matter what character is used as delimiter since it will not be written anywhere. In that case, the delimiter could be formatted as 1X and the FSML processor will ignore it during the read. For more discussion on the concept of the FSML read/write process, see Section 1.2.)

The flanking records for static text and target speed (`targspeed`) are included only to show how formatted data handling can be used with other types of elements.

Snippet from the Input Deck

```
*** Target Info ***
targlat    44-20.00-N
targlong    40-00.00-E
targspeed  12.0
```

Snippet from the FSML File

```
<FileData>
```

```
  <String name="targlat"/>
  <String name="targlong"/>
  <Integer name="targlatdeg" description="Target Latitude Degrees"/>
  <Integer name="targlatmins" description="Target Latitude Minutes"/>
  <Integer name="targlathemi" description="Target Latitude Hemisphere"/>
  <Integer name="targlongdeg" description="Target Longitude Degrees"/>
  <Integer name="targlongmins" description="Target Longitude Minutes"/>
  <Integer name="targlonghemi" description="Target Longitude Hemisphere"/>
  <Float name="targspeed" description="Target Speed"/>
</FileData>

<FileStructure>
    <!-- Target Info -->
        <StaticText>*** Target Info ***</StaticText>
        <Record ignoreOnRead="false" format="(A7,4X,I2,A1,F5.2,A1,A1)">
           <SetValue target="targlatname"/>
           <SetValue target="targlatdeg"/>
           <SetValue target="delimiter1"/>
           <SetValue target="targlatmins"/>
           <SetValue target="delimiter2"/>
           <SetValue target="targlathemi"/>
        </Record>
        <Record ignoreOnRead="false" format="(A8,4X,I2,A1,F5.2,A1,A1)">
           <SetValue target="targlongname"/>
           <SetValue target="targlongdeg"/>
           <SetValue target="delimiter1"/>
           <SetValue target="targlongmins"/>
           <SetValue target="delimiter2"/>
           <SetValue target="targlonghemi"/>
        </Record>
        <Record ignoreOnRead="false">
           <Separator ignoreOnRead="false" string="targspeed"/>
           <SetValue target="targspeed"/>
        </Record>
</FileStructure>
```

3.5 How to Process Static Text

As explained in Section 2.3.5, lines of text in the input file are processed using the StaticText element. In this example, the input file begins with four lines of text, so the StaticText tags are the first entries in the FileStructure section. Remember that the StaticText element requires no entries in the <FileData> section of the FSML file, so that doesn't appear in the snippet below.

Snippet from the Input Deck

```
***
* Run Info
* Version: 5.1.10
***
modelrun    7
clockmode   1
```

Snippet from the FSML File

```
<FileStructure>
    <!--Header for Run Info -->
        <StaticText>***</StaticText>
        <StaticText>* Run Info</StaticText>
        <StaticText>* Version: 5.1.11</StaticText>
        <StaticText>*** </StaticText>
    .
    .
    .
</FileStructure>
```

3.6 How to Process a Condition

This section presents an example of how to process a Condition element. In this case, the variable ISEA can have three possible values – 1, 2, or 3 – each of which requires different variables in the input file. Below are fragments of the Input File (one for each value of ISEA) and the FSML for reading that file (line numbers were added for reference), followed by a step-by-step explanation of how that FSML was derived. At the end is the original Fortran source code.

Input File for ISEA=1

```
!21 ISEA  NWAVES  NWSC
     1      1      0
!21 (cont) FREQW  PHASEW  AMPW  HEADW
            2.0    0.0    .1     180.0
```

Input File for ISEA=2

```
!21 ISEA  SIGWHT  TMODAL  SEAHD  SPREAD  NFREQ  NHEAD  NWSC
     2    0.027    2.75   180.0    0.0    20      1     2
!21 (cont) WSCSTP  WSCFAC = wave scaling data
           0       0.0
           50      1.0
```

Input File for ISEA=3

```
!21 ISEA  WSPEC  NWSC   (Stokes wave)
    3      5      2
!21 (cont) FREQW       PHASEW     AMPW     HEADW = single frequency Stokes wave
           2.000000    .000000    .100000  180.000000
!21 (cont) WSCSTP  WSCFAC = wave scaling data
           0        0.0
           50       1.0
```

FSML Fragment

```
     <FileData>
     ...
         <Integer name="ISEA"/>
         <Integer name="NWAVES"/>
         <Integer name="NWSC"/>
         <Float name="FREQW"/>
```

```
        <Float name="PHASEW"/>
        <Float name="AMPW"/>
        <Float name="HEADW"/>
        <Float name="SIGWHT"/>
        <Float name="TMODAL"/>
        <Float name="SEAHD"/>
        <Float name="SPREAD"/>
        <Float name="NFREQ"/>
        <Float name="NHEAD"/>
        <Integer name="WSCSTP"/>
        <Float name="WSCFAC"/>
        <Integer name="WSPEC"/>
    ...
    </FileData>
    <FileStructure>
    ...
1       <StaticText>!21 ISEA and associated variables</StaticText>
2       <Record ignoreOnRead="false">
3         <SetValue target="ISEA" />
4         <Condition check="ISEA" operator="EQ" value="1">
5           <SetValue target="NWAVES" />
6           <SetValue target="NWSC" />
7         </Condition>
8         <Condition check="ISEA" operator="EQ" value="2">
9           <SetValue target="SIGWHT" />
10          <SetValue target="TMODAL" />
11        </Condition>
12        <Condition check="ISEA" operator="EQ" value="3">
13          <SetValue target="WSPEC" />
14          <SetValue target="NWSC" />
15        </Condition>
16      </Record>
17      <Condition check="ISEA" operator="EQ" value="1">
18        <StaticText>!21 (cont) FREQW PHASEW AMPW HEADW</StaticText>
19        <Loop start="1" end="NWAVES" incr="1">
20          <Record ignoreOnRead="true">
21            <AddValue target="FREQW" />
22            <AddValue target="PHASEW" />
23            <AddValue target="AMPW" />
24            <AddValue target="HEADW" />
25          </Record>
26        </Loop>
27      </Condition>
28      <Condition check="ISEA" operator="EQ" value="2">
29        <StaticText />
```

```
30        <Record ignoreOnRead="true">
31          <SetValue target="SEAHD" />
32          <SetValue target="SPREAD" />
33          <SetValue target="NFREQ" />
34          <SetValue target="NHEAD" />
35          <SetValue target="NWSC" />
36        </Record>
37      </Condition>
38      <Condition check="ISEA" operator="EQ" value="3">
39        <StaticText>!21 (cont) FREQW       PHASEW     AMPW  ...</StaticText>
40        <Record ignoreOnRead="true">
41          <AddValue target="FREQW" />
42          <AddValue target="PHASEW" />
43          <AddValue target="AMPW" />
44          <AddValue target="HEADW" />
45        </Record>
46      </Condition>
47      <Condition check="ISEA" operator="GT" value="0">
48        <StaticText />
49        <Loop start="1" end="NWSC" incr="1">
50          <Record ignoreOnRead="true">
51            <AddValue target="WSCSTP" />
52            <AddValue target="WSCFAC" />
53          </Record>
54        </Loop>
55      </Condition>
    ...
    </FileStructure>
```

Although it is not defined in the code, a comment typically appears in the file before the value of ISEA is read. We can ensure that this comment is written to the generated file by declaring the StaticText element on line 1 in the FSML.

Next, the value of ISEA is read on line 1 in the code. Because this value is read with its own read statement, it requires moving to the next line in the file before reading. Therefore, a Record is declared with ignoreOnRead set to false on line 2 of the FSML.

The SetValue element then reads (or writes the value of ISEA). Line 2 in the code declares an if statement that corresponds to the Condition declared on line 4. If ISEA = 1, NWAVES and NWSC should be read (or written). This is done on lines 5 and 6 in the FSML.

Each possible value for ISEA requires a separate Condition statement. Lines 5 and 6 in the code indicate an iterative read of some additional parameters. Note that one value is read for each parameter during a single loop iteration. This is represented in FSML by the loop

declared on line 19. Because the values are being read with a free format, a new line for each set of parameter values is not required and should not be expected. However, for file readability it is desirable to have each set of parameter values on a separate line. Hence the reading (or writing) of the parameter values is performed on lines 21–24 in the FSML within the record on line 20, which has ignoreOnRead set to true. Why was the loop not defined inside the condition on line 4? Because when the file is written, it is desired, in this case, to have these values start on a new line. This can only be achieved with a record whose ignoreOnRead is set to true.

The condition on line 4 was defined within a record because ISEA should always be read (or written), but other values should be read from the same line depending on the value of ISEA. Because a record cannot contain another record, it is necessary to declare the loop outside of the condition on line 4. Finally, note that in the code, the WSCSTP and WSCFAC arrays are read for all values of ISEA greater than 0. Another loop is therefore declared on line 49 in the FSML for reading these values.

Fortran Code Fragment

```
1      read(uin,*,err=901,end=900) isea
2      if (isea.eq.1) then
3      backspace(uin)
4      read(uin,*,err=901,end=900) isea,nwaves,nwsc,
5     .        (freqw(min(i,maxwav)),phasew(min(i,maxwav)),
6     .       ampw(min(i,maxwav)),headw(min(i,maxwav)),i=1,nwaves),
7     .        (wscstp(min(i,mxspec)),wscfac(min(i,mxspec)),i=1,nwsc)
8      elseif (isea.eq.2) then
9       backspace(uin)
10      read(uin,*,err=901,end=900) isea,sigwht,tmodal,
11    .        seahd,spread,nfreq,nhead,nwsc,
12    .        (wscstp(min(i,mxspec)),wscfac(min(i,mxspec)),i=1,nwsc)
13     elseif (isea.eq.3) then
14      backspace(uin)
15      read(uin,*,err=901,end=900) isea,wspec,nwsc,
16    .        freqw(1),phasew(1),ampw(1),headw(1),
17    .        (wscstp(min(i,mxspec)),wscfac(min(i,mxspec)),i=1,nwsc)
18     endif
```

3.7 Putting It All Together: Describing a Legacy Application Input File

As pictured in Figure 1 in Section 1.2, each input and output file for a legacy application has a corresponding FSML file that describes its contents. This section shows a typical legacy input file and the FSML file that describes it.

Legacy Input File

```
! Data file for lethality evaluator
1.22  , TNT Equivalence Factor
! ****************************************
! Fuzing option
! 0 - contact detonation
! 1 - standoff detonation
  0
! ****************************************
! if  fuzing option = 0, specify:
0.3, Torpedo Diameter (m)
1700    , Warhead density (kg/m3)
3       , number of noselength data points
! noselength data
! Diameter(m)   Nose Length(m)
  .1524         .254
  .3048         .3048
  .5334         .4572
! else specify standoff distance in m
! 12.5
 4       , Number of Pkill data points
! Pkill Data
! sqrt(TNT eq(kg))/(detonation distance(m))  Pkill
  0                                          0
  8.522                                      .1
  9.882                                      .5
  11.172                                     .9
```

Corresponding FSML File

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE FileTypeSpecification SYSTEM
"file:/F:/ONR/UWDO/Codes/letheval/fsml/FSMLv2.7.dtd">
<FileTypeSpecification lineCommentStart="!" ignoreBlankLines="false">
  <FileData>
    <Float name="TNT_Equivalence_Factor"/>
    <Integer name="Fuzing_Option"/>
    <Float name="Torpedo_Diameter"/>
    <Integer name="Warhead_Density"/>
    <Integer name="NoseLength_Data_Points"/>
    <Array name="Diameter" ndims="1" dims="10" baseType="Float"/>
    <Array name="NoseLength" ndims="1" dims="10" baseType="Float"/>
    <Integer name="Pkill_Data_Points"/>
    <Array name="TNT_EQ-Detonation_Dist" ndims="1" dims="10"
           baseType="Float"/>
    <Array name="Pkill" ndims="1" dims="10" baseType="Float"/>
  </FileData>
  <FileStructure>
    <StaticText>! Data file for lethality evaluator</StaticText>

    <Record ignoreOnRead="false">
      <SetValue target="TNT_Equivalence_Factor"/>
    </Record>

    <StaticText>! ****************************************</StaticText>
    <StaticText>! Fuzing option</StaticText>
    <StaticText>! 0 - contact detonation</StaticText>
    <StaticText>! 1 - standoff detonation</StaticText>

    <Record ignoreOnRead="false">
      <SetValue target="Fuzing_Option"/>
    </Record>

    <StaticText>! ****************************************</StaticText>
    <StaticText>! if fuzing option = 0, specify:</StaticText>

    <Record ignoreOnRead="false">
      <SetValue target="Torpedo_Diameter"/>
    </Record>

    <Record ignoreOnRead="false">
      <SetValue target="Warhead_Density"/>
    </Record>
```

```
    <Record ignoreOnRead="false">
      <SetValue target="NoseLength_Data_Points"/>
    </Record>

    <StaticText>! noselength data</StaticText>
    <StaticText>! Diameter(m)   Nose Length(m)</StaticText>

    <Loop loopVariable="i" start="0" end="2" incr="1" doOnce="true">
      <Record ignoreOnRead="true">
        <AddValue target="Diameter"/>
        <AddValue target="NoseLength"/>
      </Record>
    </Loop>

    <StaticText>! else specify standoff distance in m</StaticText>
    <StaticText>! 12.5</StaticText>

    <Record ignoreOnRead="false">
      <SetValue target="Pkill_Data_Points"/>
    </Record>

    <StaticText>! Pkill Data</StaticText>
    <StaticText>! sqrt(TNT eq(kg))/(detonation distance(m))</StaticText>

    <Loop loopVariable="i" start="0" end="3" incr="1" doOnce="true">
      <Record ignoreOnRead="true">
        <AddValue target="TNT_EQ-Detonation_Dist"/>
        <AddValue target="Pkill"/>
      </Record>
    </Loop>

  </FileStructure>
</FileTypeSpecification>
```

4 FSML Document Type Definition (DTD)

This section presents the FSML DTD. The DTD defines the grammar for FSML and provides a means to validate an FSML document's structure and syntax. However, an FSML document that is valid according to the DTD is not necessarily a valid FSML document. The DTD does not, for example, identify parameter references that are made before the value of the referenced parameter is set. It is the responsibility of the FSML processor to perform additional validation on the document.

```
<!--Header
FSML.dtd (File Specification Markup Language)
File Description:       XML DTD for representing a text file's structure
Version:                2.9
Date of Last Revision:  February 9, 2004
Contact:                Eric Wood(wood@ship.saic.com)
Namespace:              none
Notes:                  This revision provides the ability to set whether
                        blank values should be ignored or reported as an error
-->

<!ENTITY % FILE_DATA_CM
"
(Integer | Float | String | Array)+
">

<!ENTITY % DATA_CM
"
   EMPTY
" >

<!ENTITY % DATA_ATT
"
   name         ID    #REQUIRED
   description CDATA #IMPLIED
   units       CDATA #IMPLIED
   group       CDATA #IMPLIED
" >

<!ENTITY % FILE_STRUCTURE_CM
"
(Record | Loop | Condition | StaticText)+
">
```

```
<!ENTITY % OPS
"
(GT | LT | EQ | NE)
">

<!ENTITY % RECORD_CM
"
SetValue | SetDims | SetSize | AddValue | Loop | Condition | Separator
">

<!ENTITY % OPERATION_ATTS
"
target   IDREF   #REQUIRED
">

<!ENTITY % MUTATOR_ATTS
"
%OPERATION_ATTS;
">

<!ELEMENT FileTypeSpecification (FileData,FileStructure)>
<!ATTLIST FileTypeSpecification
   type                CDATA          #IMPLIED
   description         CDATA          #IMPLIED
   lineCommentStart    CDATA          #IMPLIED
   blockCommentStart   CDATA          #IMPLIED
   blockCommentEnd     CDATA          #IMPLIED
   ignoreBlankLines    (true | false) "true"
   delimiters          CDATA          #IMPLIED
   ignoreMissingValues (true | false) "false">

<!ELEMENT FileData %FILE_DATA_CM;>

<!ELEMENT Integer %DATA_CM; >
<!ATTLIST Integer %DATA_ATT; >

<!ELEMENT Float %DATA_CM; >
<!ATTLIST Float %DATA_ATT; >

<!ELEMENT String %DATA_CM; >
<!ATTLIST String %DATA_ATT; >

<!ELEMENT Array %DATA_CM; >
<!ATTLIST Array %DATA_ATT;
   ndims     CDATA                      #REQUIRED
```

```
   dims      CDATA                       #REQUIRED
   baseType  (String | Integer | Float) #REQUIRED >

<!ELEMENT Dim EMPTY >
<!ATTLIST Dim
   size      CDATA #IMPLIED >

<!ELEMENT FileStructure %FILE_STRUCTURE_CM;>

<!ELEMENT StaticText (#PCDATA) >

<!ELEMENT Record (%RECORD_CM;)+>
<!ATTLIST Record
   ignoreOnRead       (true | false) #REQUIRED
   format             CDATA          #IMPLIED
   ignoreFormatOnRead (true | false) #IMPLIED
>

<!ELEMENT SetValue EMPTY >
<!ATTLIST SetValue
   %MUTATOR_ATTS;
>

<!ELEMENT AddValue EMPTY >
<!ATTLIST AddValue
   %MUTATOR_ATTS;
>

<!ELEMENT SetDims EMPTY >
<!ATTLIST SetDims
   %OPERATION_ATTS;
   dims       CDATA   #REQUIRED
>

<!ELEMENT SetSize EMPTY >
<!ATTLIST SetSize
   %OPERATION_ATTS;
   dimIndex   CDATA   #REQUIRED
   size       CDATA   #REQUIRED
>

<!ELEMENT Loop (%RECORD_CM; | Record)+>
<!ATTLIST Loop
   loopVariable CDATA          #IMPLIED
   start        CDATA          #REQUIRED
```

```
   end          CDATA          #REQUIRED
   incr         CDATA          #REQUIRED
   check        CDATA          #IMPLIED
   operator     %OPS;          #IMPLIED
   checkValue   CDATA          #IMPLIED
   stopOnEof    (true | false) #IMPLIED
   doOnce       (true | false) #IMPLIED
>

<!ELEMENT Condition (%RECORD_CM; | Record | StaticText)+>
<!ATTLIST Condition
   check        CDATA          #REQUIRED
   operator     %OPS;          #REQUIRED
   checkValue   CDATA          #REQUIRED
>

<!ELEMENT Separator EMPTY>
<!ATTLIST Separator
   ignoreOnRead (true | false) #REQUIRED
   string       CDATA          #REQUIRED
   repeat       CDATA          #IMPLIED
>
```