

A New Look at the Standardization of Scientific Data

Ryan Skar
Computer Science Department
Augsburg College
Minneapolis, MN 55454
skar@augsborg.edu

This work was supported by NASA grant NNG05GE18G.

Abstract

Scientists in the field of Space and Solar Physics have in the last few years accumulated several terabits of data. Data collected from these sites are stored at several places throughout the world, all in different formats and with different modes of access. In order for this data to be useful there must be a standard way to define the data available. The SPASE consortium has been working to make all these different data available to the scientific community.

A uniform set of terms has been defined for how data are to be described by SPASE. This was accomplished using an XML schema. The next step is to make a uniform way to store, add to and search the different sets of data from anywhere in the world. This paper will explain the software to accomplish this.

1 Introduction

Data has been around for as long as people have tried to quantify the world around them. Data can be anything from someone's shoe size to the change in the magnetic field along the vertical axis. It is through using data that so many accomplishments have been made, but with this data comes the problem of standardization. The problem comes from the fact that when data are taken in, the purpose is to represent something else that is found in the environment. Considering that data simplifies the various characterizations which the object possesses (e.g., height, weight, color), there can be some confusion as to what data means. Is the length of an object one foot or two feet? Whose feet? One of the earliest measurements, the foot, dating back to Babylonian times, was modeled after a human foot [1]. This could be confusing as not everyone's foot is the same size. This is only one of the many examples that led to the need for data standardization.

In the 21st century, standard units exist to measure the data that is collected. These units help to eliminate some of the confusion of what data means, but there is more to understanding data than the units used when recording data. One must also think of the sample rate at which data is taken. A measurement can be taken only once or a series of times. Also how accurate do you want the measurement to be? The height of an object could be taken in m, cm, mm, etc. These different decisions can change the look of data. After all the data are collected they must also be stored. This could be accomplished in a text file, a binary file or maybe even a graph. These decisions have been made by scientists throughout history.

In the area of Space and Solar Physics the need for easy access to data is growing. With several different types of instruments from locations both on the earth's surface and in space, finding the data needed has become a difficult task. Each group uses a slightly different way of obtaining and storing the data and there is no way of knowing what data is stored at a specific site. Also more of the research in the area of Space and Solar Physics is requiring several different sources of data. This increases the need for a standard way of dealing with data.

1.1 Previous Attempts at Standardization

Due to the fact that the data standardization problem has been around for a while there have been several attempts to create a standard way for storing data. The goal was to make data readable even if nothing was known about the data before hand. Most attempts tried to make the data files so that they were self describing. A header would be added to the beginning of the file to describe the data that followed. The way that each standard made the header and stored the data set them apart. Some of the more common data standards are: Common Data Format (CDF), Flexible Image Transport System (FITS), and Hierarchical Data Format (HDF). These different attempts have had varying

degrees of success and are still used today, but no one standard has been accepted in all the scientific community.

The Common Data Format was created by NASA to help with their Climate Data System [2]. It created an abstract way of accessing, editing and storing data. The goal was to format the data in such a way that it could be manipulated by a universal program. The standard consisted of a header which contained global descriptions such as title, documentation, and modification, followed by variable specifications such as field name, min/max values and order. The data was then stored in an array format following the header. The problem with the CDF is that it was difficult to describe the data in such a way that a universal program could manipulate it accurately. Scientists usually left the data in the format in which the instruments collected it. However, these formats were usually different than the CDF standard. In order to conform to the CDF standard all the data had to be converted. This process was very time consuming and required writing a program to convert from the previous format to the CDF format. In many cases it was easier to write programs to manipulate the data in the current format, then convert it to the CDF format and then manipulate the data with the universal program.

The Flexible Image Transport System was similar to the CDF in that it used headers to describe the data [3]. The difference was that FITS used a more concrete system for describing data and how the data was stored. The different portions of the file were divided up into HDUs (Header/Data Units). Each HDU consists of a header unit that describes the data contained in the HDU. Instead of the header being dynamic it was restricted to having a fixed length of 2880 bytes. The header also had five required keywords that must appear in each file. The data in FITS was not limited to being stored in just ASCII format; there were also image and binary extensions for storing the data. The option of a binary format made it more versatile than CDF, as large amounts of data could be condensed when using a binary format. The fact that the headers are more concrete than the CDF format makes it easier to create a universal data manipulator, but the problem remains of having to convert the data to conform to the FITS standard.

The Hierarchical Data Format mixes the CDF and FITS standards [4]. Like the FITS standard the header is of a fixed length with the first 46 bytes identifying the file. The next part of the file then consists of data descriptor blocks which describe how the data can be referenced in the file. Each data descriptor block is 12 bytes long which was divided up into four parts: a two byte tag, a two byte reference number, a four byte offset, and a four byte length. Following the descriptor blocks the data is stored in a binary data object. This allows for any part of the data to be accessed individually as each variable contains an offset where the data are stored. The main drawback of the HDF is the same as CDF and FITS: the data still needs to be changed to conform to the proper format.

2 SPASE Data Standard

The Space Physics Archive Search and Extract (SPASE) consortium is made up of an international team of Space and Solar physicists and information scientists [5]. The

SPASE consortium has attempted to address the problem of data standardization from a new direction. Instead of trying to convert the data into a standard form from which it can easily be read using a global compiler, SPASE attempts instead to create a standard set of terms from which data can be defined. Then using these standard ways to define data it would be easy to catalog the data that is available from around the world. SPASE is concerned with defining Space and Solar Physics resources. This paradigm follows closely from that of a "Virtual Observatory." The basic idea of a Virtual Observatory is an archive of data and software tools to modify the data available through the web. So far this paradigm has been successfully applied using the Planetary Data System (PDS) for Planetary Science and the International Virtual Observatory Alliance (IVOA) for Astronomy and Astrophysics. SPASE attempts to expand on this model by providing a database of links to the data and software instead of storing all the data and software in the database. This allows the database to be significantly smaller while still fulfilling the same purpose. Once the links have been defined it would be an easy matter to retrieve the data using simple searches and then use the programs that have already been developed to manipulate the data.

The focus of the SPASE model is to define products, similar groups of data, image and plot files. The files can be generalized in this way because they share the same purpose and format. The definition of products does not need to be exacting as it does not need to describe the data completely; the definition only needs to describe the basic information about the data, enough to understand what the data represents, and to call the data. This generalization makes it easy to search for data with simple criteria such as source, measurement description, and time collected without having to worry about all the specifics.

In order to define the vocabulary that is to be used to describe data, the SPASE consortium examined the current existing models and from them took the vocabulary needed to accurately describe the data. They found that the product definitions could be split into Numerical, Data Display and Catalogue. These types could then be linked to the resource types Observatory, Repository, Instrument, and Person which would give a complete description of the entire product. In order to define this standard, the SPASE consortium used eXtensible Markup Language (XML). Using XML schema they were able to create a hierarchical representation of what parts needed to appear in each of the main types. The XML schema allowed them to show, using attributes, which elements were required and which were optional as well as which could appear more than once as seen in Figure 2.1.

The schema also controlled what would be allowed in any definition of data, filtering all the parts that were not applicable to the current type. For example, start and stop dates are not relevant when describing a person. Only the nodes that are children of a product node would be used to describe the data. The rest would be skipped over. As seen in Figure 2.1 when a person node is selected a large part of the document is omitted.

```
...
<xs:element name="PERSON">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="RESOURCE_ID"/>
      <xs:element minOccurs="0" ref="PERSON_NAME"/>
      <xs:element ref="ORGANIZATION_NAME"/>
      <xs:element minOccurs="0" ref="ADDRESS"/>
      <xs:element minOccurs="0" maxOccurs="unbounded" ref="E-MAIL"/>
      <xs:element minOccurs="0" maxOccurs="unbounded" ref="PHONE_NUMBER"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

```

</xs:element>
...
<xs:element name="ORGANIZATION_NAME" type="xs:string"/>
<xs:element name="ADDRESS" type="xs:string"/>
<xs:element name="E-MAIL" type="xs:string"/>
<xs:element name="PHONE_NUMBER" type="xs:string"/>
<xs:element name="RESOURCE_ID" type="xs:string"/>
...

```

3 Implementation

Now that a standard data definition is created, the next step is to implement the standard. There are several different ways that this could be accomplished. Each group of scientists could apply the standard to their data and post those definitions on their web page. However, this would not move them very far from where they were before the standard was developed. The data would be defined in a standard way, but it would be for the most part inaccessible to the rest of the scientific community. There would be no way to know if there were Antarctic magnetometer data available for August of 2004 unless each site was searched individually. This process would take a significant amount of time and the chances of finding all the data available would still be small. This means that there must be some way of cataloging the data that is available at one or many central locations. If this was done, different data definitions could be added into these locations either manually or automatically. It would then be an easy matter to go to one of these locations and search for available data which meets the described specifications. The system needed for this must be available over the web with no downloads required to use. In light of these goals, Java Servlets appeared to be an appropriate architecture to use in creating one of these databases.

A servlet is a Java based platform to create web based applications. The servlet still has all the functionality of a regular Java program as well as the ability to send and receive Hypertext Transfer Protocol (HTTP) requests. Servlets also have access to a host of Java Application Program Interfaces (API) including the Java Database Connectivity (JDBC) API, which allows the servlet to connect to databases. The outline for the process can be seen in Figure 3.1.

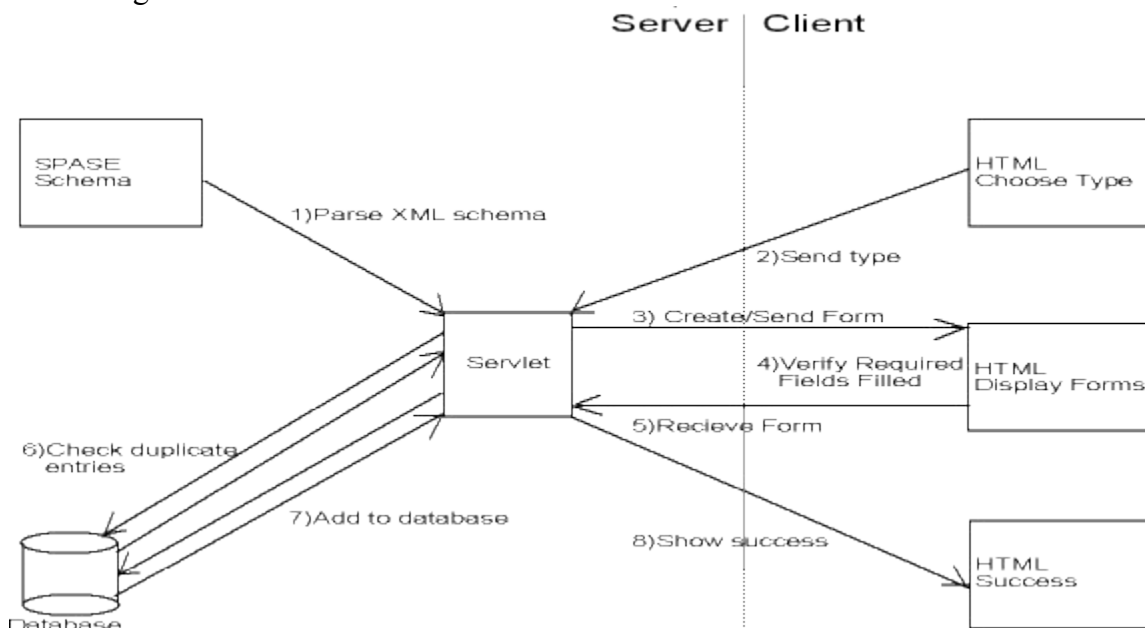


Figure 3.1 Flow chart for the web interface.

3.1 Validation program

The specifications for the SPASE data standardization are stored in an XML document, so the first step in getting the interface to work is to load and parse the XML file into a format that is Java readable. There are three standard ways of parsing an XML document: Simple API for XML (SAX), Document Object Model (DOM) or JDOM [6]. These different standards were developed by different groups in an effort to make a standard way of parsing XML. SAX, an open source effort, reads through the XML file linearly, so it would be difficult to build an interface that uses different nodes more than once as they would have to be stored manually then looked up when needed again. DOM, the W3C standard, and JDOM, another open source effort, use a tree like structure, making it easier to reach nodes in a non-linear fashion. This means that both methods would work, but JDOM had the added benefit of containing concrete methods accessing the nodes and their attributes. This made the process of traversing the node structure easier, which is why JDOM was chosen over the others. Once a method had been chosen it was an easy matter to parse the XML file.

After the XML schema file was parsed, it could be used to construct Hypertext Markup Language (HTML) forms. These forms could then be used to add data specifications to a database. To do this the servlet must know a place to start in the node structure. Due to the way the schema is set up there are several different data types for which the specification is defined: Catalog, Display Data, Instrument, Numerical Data, Observatory and Person. Based on the type of data the schema would be specified using different starting points. To find out which type of data specification is to be created, a small HTML form was created to offer the client a choice and then return that value to the servlet. This allowed the servlet to pick a starting point in the array and start building the forms for the current specification.

3.2 Form Generation

Once a data type has been picked through the simple HTML form and passed back to the servlet, the servlet can begin traversing the parsed XML schema. To do this the servlet must go through the tree in a depth first traversal and grab on to the relevant data found at each node. The servlet must also follow references to other nodes as they appear in the traversal.

Traversing through the tree was done using two recursive methods. The first method worked through each node and accumulated the needed data, such as node name, node type and node attributes. The method also visits the children of the node in a depth first fashion as seen in Figure 3.2.

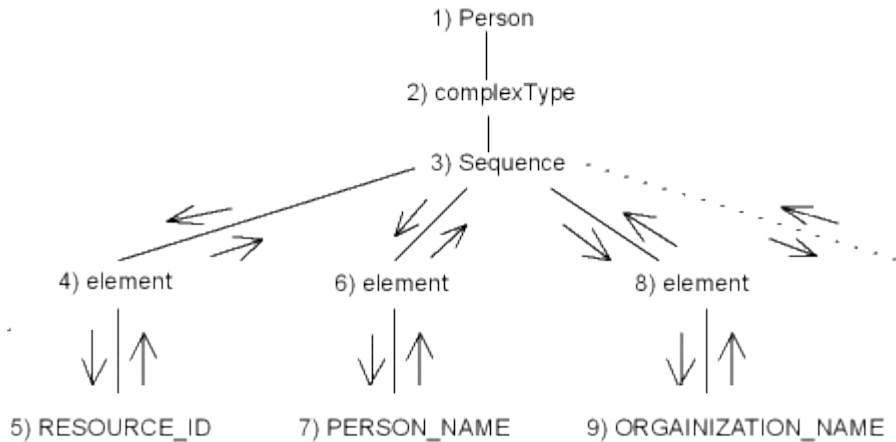


Figure 3.2 Example of a Depth First Search.

This allowed the code for the HTML forms to be created during the traversal. The second method is used to gather the attributes of each node and to follow any references from one node to another. The two methods work hand in hand to identify the nodes where data specifications need to be added and create a form that can be used to input data specifications. These methods also keep track of the element name of each form created so that it can later be used to get the values back from the HTML after it has been modified by the user. To accomplish this, each time a leaf node is accessed, its name is added to the end of a vector. Also since the node may be accessed more than once at different parts of the tree, some code must be implemented to differentiate between the instances of the node each time it is called. To accomplish this, the parent nodes are also stored as part of the name. For example, the node named email would be [root].[parent].email with each ancestor separated by a period. This allows for multiple instances of a form element to occur without confusing which belongs to which. The code is displayed in Figure 3.3.

```

...
Vector formNames = new Vector(); //creates the vector to store the node names
parent = req.getParameter("DataType"); //gets starting type (root value)
...
//When a node is a non-leaf node the following code is executed.
parent=parent+formName+".";
...
//When node is a leaf node the following code is executed.
String id = parent+formName;
formNames.add(id);
...

```

Figure 3.3 Code for accumulating data fields.

After the form has been created it is then passed by an HTTP request back to the client where the form then acts the same as an HTML form. The client does not notice any difference between the first form, where they chose the data type and this one, where they

can add specifications, but the first was static while the other was created dynamically by the servlet. The form for the data specification looks something like Figure 3.4 depending on what data type was selected.

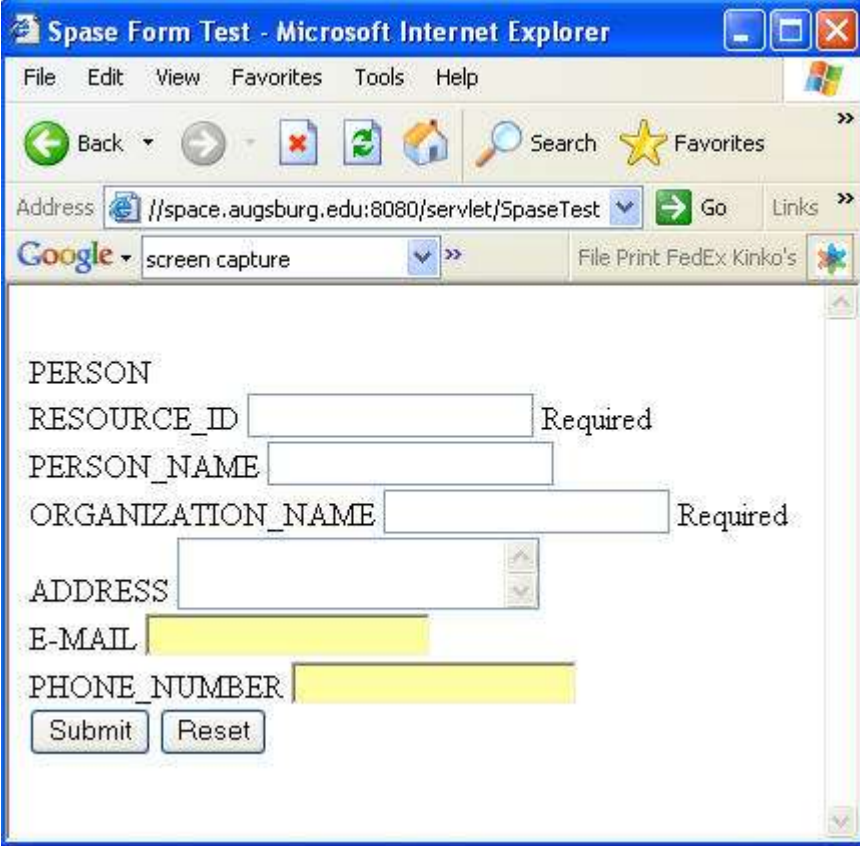
A screenshot of a Microsoft Internet Explorer browser window titled "Spase Form Test - Microsoft Internet Explorer". The address bar shows "http://space.augsburg.edu:8080/servlet/SpaseTest". The search bar contains "screen capture". The main content area displays a form with the following fields: "PERSON_RESOURCE_ID" (text input, marked "Required"), "PERSON_NAME" (text input), "ORGANIZATION_NAME" (text input, marked "Required"), "ADDRESS" (text input), "E-MAIL" (text input, highlighted in yellow), and "PHONE_NUMBER" (text input, highlighted in yellow). At the bottom of the form are "Submit" and "Reset" buttons.

Figure 3.4 Form for data specification.

3.3 Form Validation

Once the client has received the form, the fields can be edited to match the data. After the form is complete, it is sent back to the servlet where the values are processed. To make sure that all the necessary fields were filled, the form needs to undergo some form of verification. This could either be done on the client side, the server side or both. To make the validation streamlined, the form validation was done on the client side with the remaining validation done on the server side. To do the client side validation, a few JavaScript functions were written to make sure that all required elements were filled. These functions are generic; to save lines of code, they were placed in a separate file and accessed from the servlet. The required fields were found during the creation of the forms and it was therefore easy to add the calls to the JavaScript functions into those methods using the `onblur()` operation as seen in Figure 3.5.


```

if(req)
{
    // Prints out a text input to the HTML which cannot be left blank.
    out.println("<input type='text' id='"+id+"' name='"+id+"
                "' ONBLUR=present(this,'2"+id+"');>");
    //Creates a span that is later used to print error messages if necessary.
    out.println("<span id='2"+id+"'>Required</span>");
}
else
    // Prints out a text input field to the HTML with no validation.
    out.println("<input type='text' name='"+id+"'>");

```

Figure 3.5 Code to produce HTML form fields.

The method checks the value of the element to see if it is empty and if so writes an error message to the HTML page. The result of the code is client-side element by element validation as seen in Figure 3.6.

CATALOG

RESOURCE_ID Error: value required

RESOURCE_HEADER

RESOURCE_NAME Required

Figure 3.6 Client Side Validation.

To ensure that all the required fields are filled in, the entire form is checked again before submission. In order to confirm that fields (e.g. phone number, email) contain a particular structure, other JavaScript functions can be implemented in the same way. Once the form has passed the client side validation it is passed back to the servlet.

After the servlet receives the form via the post request, it takes all the form values and checks to see that certain values, the IDs for example, are not duplicated. The servlet does this by creating a connection to the MySQL database using Connector/J a native driver for JDBC (Java Database Connectivity) [7]. Using this connection the servlet then compares the values from the current form with those already in the table. If there is a duplicate entry, the servlet will send the form back to the client and prompt them to choose a different value for the duplicated entry. This keeps the database free of duplicated data specifications. If the form passes the server side validation the servlet adds the form fields to the MySQL database using the insert command. Now the data description is available to anyone checking the database. If the values have been added into the database correctly the servlet will send a success message back to the client along with the information they just added.

4 Benefits

The main advantage of the SPASE standard is that there is no changing of how data are stored. All that is needed is to add a data specification to the database so that the users will be able to find it with a search and link to where it is stored. There is no converting of the data to meet the new standard. This makes it much easier to use than some of the other formats such as CDF, which requires that data be wrapped in a self describing header. SPASE does not try to change the data into a new form, but only to list the data that is available. Another benefit of this implementation is there is no special software required by the client, so it is easily usable from any computer with Internet access. The fact that the server side is made up of a XML file and a servlet makes it extremely portable. The MySQL database need not be local, but can instead be anywhere in the world allowing several of these servlets to feed into one database. Also as the standard grows and changes, the servlet itself will need little or no modification since all the rules are contained in the XML file. This means that to update the servlet to a new version of the schema all that needs to be done is to destroy and reinitialize the servlet context. This will load the new XML schema, which makes updating nearly automatic. Due to the fact that the servlet is written in Java it will be easy to implement new functionality to the servlet by adding classes which can be easily referenced from the servlet.

5 Future Work

Now that an interface has been created to define and store data descriptions, a method of searching these definitions must be established. The SPASE standard can describe data in both a general and a specific way. Therefore, the methods for searching the data can range from simple field based searches to complex multi-field queries. The focus is on the ease in which a search is conducted rather than on what methods are used. The search should seamlessly find data meeting the client's specifications from local and remote sites automatically. Data found from a remote site should look identical to data found locally. The transparency of data regardless of its source is the ultimate goal of SPASE. To accomplish this goal, new methods to share data between databases must be created and automated. Once this is accomplished, data will be available from around the world regardless of where the search originated.

References

- [1] O'Connor, J J and Robertson, E F. "The history of measurement." *JOC/EFR*. 2003. HTML. Available: [HTTP://www-history.mcs.st-andrews.ac.uk/HistTopics/Measurement.HTML](http://www-history.mcs.st-andrews.ac.uk/HistTopics/Measurement.HTML)
- [2] Space Physics Data Facility. "CDF User's Guide." *Common Data Format*. 2006. pdf. Available: <ftp://nssdcftp.gsfc.nasa.gov/standards/cdf/doc/cdf31/cdf31ug.pdf>
- [3] High Energy Astrophysics Science Archive Research Center. "A Primer on the FITS Data Format." *Flexible Image Transport System*. 2004. HTML. Available: [HTTP://fits.gsfc.nasa.gov/fits_primer.HTML](http://fits.gsfc.nasa.gov/fits_primer.HTML)

- [4] National Center for Supercomputing Applications. "HDF4 User's Guide." *Hierarchical Data Format*. 2003. pdf. Available: ftp://ftp.ncsa.uiuc.edu/HDF/HDF/Documentation/HDF4.2r0/HDF42r0_UserGd.pdf
- [5] The SPASE consortium. "Space and Solar Physics Data Model." *Space Physics Archive and Extract*. 2006. pdf. Available: <HTTP://www.igpp.ucla.edu/spase/data/makedoc.php>.
- [6] McLaughlin, Brett. Java & XML 2nd Edition. California: O'Reilly & Associates, Inc. 2001.
- [7] Khan, Faisal. "A Connecting to a MySQL Database using Connector/J JDBC Driver." *Stardeveloper*. 2003. HTML. Available: <HTTP://www.stardeveloper.com/articles/display.HTML?article=2003090401&page=1>