JXPL: An XML-based Scripting Language for Workflow Execution in a Grid Environment

Carla S. Hunt and Clayton S. Ferner
Department of Computer Science
University of North Carolina at Wilmington
cferner@uncw.edu

Jeff L. Brown
Department of Mathematics
University of North Carolina at Wilmington
brownj@uncw.edu

Abstract

JXPL is a new functional scripting language that uses XML syntax. JXPL is intended to be a workflow language that easily interacts with web and grid services. There are built in web and grid clients that can communicate with a variety of services. This facilitates the creation of workflow editors and other applications designed for grid environments and leaves application developers free to concentrate on the implementation of the user interface. In this paper we give our motivation for JXPL and describe JXPL so that others may use the language. To aid our discussion, we introduce the grammar for JXPL using Extended Backus Naur Form (EBNF). We also include examples and detail predefined functions. We conclude with current and future work.

1. Introduction

Grid computing has received much attention in the research community in recent years. Grids offer users access to computing resources such as processing cycles, software, data, storage, etc. in a seamless environment. Grids also offer universities, corporations, and other organizations a way to make better use of under-utilized resources.

Since grid computing is in its early stages of development, the user interfaces are still rather crude. Most grid software uses command-line interfaces. Two new areas of research related to grids are portals and workflow editors. Portals are web-based interfaces that allow users to submit jobs to a grid and monitor grid resources. Workflow editors allow users to create custom workflows of computation that may include computation to be performed in a grid environment.

GridNexus is one such workflow editor. GridNexus uses a scripting language called JXPL to describe the workflow, which can be executed by an interpreter. JXPL features generic web and grid clients. This makes it easy for applications such as GridNexus to interact with web and grid services.

The purpose of this paper is to describe JXPL in precise terms so that others will be able to develop JXPL scripts. JXPL is actually not intended to be a language written by programmers, but rather generated by applications such as GridNexus. We wish to encourage developers to create applications that generate JXPL as a means of executing scientific workflows capable of interacting with grid and web services. This paper should serve as a reference to interested readers desiring to create new applications that can produce JXPL.

This paper is organized as follows: section 1.2 explains why it is important for there to be a language like JXPL, section 2 is an overview of the Lisp language, particularly the characteristics that have been most influential on the design of JXPL, section 3 describes the role of XML, section 4 provides the grammar for JXPL using Extended Backus Naur Form and specifics about some functions, and section 5 is future work and conclusions.

1.1 Motivation for JXPL

The flexible and extensible design of the JXPL scripting language has been instrumental in the creation of GridNexus, a powerful, graphical workflow editor for web and grid services. The JXPL language provides abstraction capabilities, support for recursion, and simplistic data structures that meliorate the difficulties associated with executing complex, graphical workflows. One of the most important benefits of using JXPL is that it separates the Graphical User Interface (GUI) from the underlying execution model. There are three reasons why this is important. First, this simplifies the implementation of the GUI. Second, this allows other tools besides GridNexus to create JXPL and therefore take advantage of its features. Third, the execution of the workflow can be performed off-line by other processors than the one interacting with the user. This allows for batch mode processing as well as parallel execution, which we discuss in sections 4 and 5.

JXPL has functions that operate as generic web and grid clients. From the perspective of the GridNexus user, there is a seamless view of a variety of geographically independent resources such as grid services, web services, data sources, and applications. The combination of a

graphical interface and predefined functions for web and grid service clients makes it possible for a novice user to create workflows that can represent fairly complex sets of tasks.

Furthermore, each JXPL interpreter has a Uniform Resource Identifier (URI). The prefix of the URI indicates the protocol used to communicate with that interpreter; that is, whether it is implemented locally, as a grid service, or as simply a remote application [1]. This gives users the flexibility to determine where units of work are computed.

Finally, JXPL programs can be used to manage Grid Data Service interactions. Open Grid Services Architecture – Data Access and Integration (OGSA-DAI) offers a special type of Globus Toolkit 3 (GT3) service called a Grid Data Service (GDS). For example, a GDS that queries a database can deliver its output to another GDS that transforms the data in some way and then delivers the result to GridFTP, or a file system, or even another service. An OGSA-DAI XML script determines the actions of a GDS. Creating a GDS for accessing an existing data source such as a relational database, an XML data source, or a file system is simply a matter of editing the OGSA-DAI configuration files. However, managing the interaction *between* Grid Data Services is complicated.

Consider the case where there are two GDSs, a *source* and a *sink*. The output of the source is to be delivered to the sink. The steps required to manage this interaction are:

1. Create the sink and store its handle.
2. Start the sink in its own thread waiting for input.
3. Create the XML script to control the source, incorporating the handle of the sink.
4. Start the source service.

This sequence of steps is easy to implement in Java, and the OGSA-DAI Application Programming Interface (API) provides many helper classes to aid in creating the necessary XML scripts. However, an *interactive* workflow engine must generate code dynamically, which is not easy to do with Java. JXPL can manage the interaction *between* Grid Data Services. In fact, some of JXPL's features, such as multi-threading, were added to make this possible.

2. Lisp

JXPL is a functional language with a Lisp-like design. Lisp is the oldest and most widely used functional programming language. Although traditionally associated with the field of Artificial Intelligence, Lisp has also been used with a high level of success in other domains. *Yahoo! Store*, one of the most successful e-commerce software applications in use today was written in Common Lisp by Paul Graham [2]. In this section we examine Lisp's power and simplicity, features that inspired JXPL's design.

One of the most appealing features of Lisp is the simplicity of its data structures: everything is either an Atom or a List. Atoms are primitive data types such as numbers, strings, and variables. Lists contain Atoms or other Lists, which allows for nesting of data structures. In both languages, the list data object is all-inclusive, meaning it can be used to define programs, data structures, and functions.

To illustrate the simplicity of the language, Figure 1 below contains a short grammar for the Lisp language using Extended Backus Naur Form (EBNF) notation. EBNF is a metalanguage used to describe the rules for a given language. We refer the reader to [4] for specifics on EBNF. We use the convention that non-terminals are capitalized, terminals are lowercase, **x**|**y** means one of either **x** or **y**, **{x}** means zero or more occurrences, and **[x]** means zero or one occurrence. For simplicity, we have omitted the rules for the non-terminals SYMBOL, NUMBER, STRING, and CONSTANT.

S-EXPR	→	ATOM | LIST | ε
LIST	→	({S-EXPR})
ATOM	→	SYMBOL | NUMBER | STRING | CONSTANT

Figure 1. Extended Backus-Naur Form (EBNF) for the Lisp Language

We read the grammar in Figure 1 to mean that the syntactic structures of the Lisp language are as follows: 1) S-Expressions are Atoms, Lists or nothing 2) Lists are zero or more S-Expressions surrounded by parentheses, and 3) Atoms are symbols, numbers, strings, and constants. It is also evident from the grammar that nesting is a natural part of the language structure.

3. Extensible Markup Language (XML)

JXPL scripts are written in XML. The parentheses of Lisp are replaced by more descriptive XML tags. Using XML to describe JXPL provides advantage in both form and function. JXPL uses the XML 1.0 specification, which is part of a widely used international standard [5]. This means JXPL is compatible with XML-aware programs, protocols, and parsers. For example, the Simple Object Access Protocol (SOAP) is the protocol used to transmit messages between grid and web services. Since SOAP is written in XML, JXPL integrates nicely with SOAP.

Both Lisp and XML are languages whose readability is poor. However, JXPL was intended to be generated by programs rather than programmers, and the XML format makes it easy for programs to write JXPL scripts. The result of executing a JXPL script is also XML.

Since all objects in the JXPL language are represented by XML elements, JXPL objects can be treated as if they are the same data type. This is analogous to a purely functional Lisp dialect in which all objects are essentially typeless. In either case, the power of the functional programming paradigm is derived from the high level of abstraction that is achieved when object typing is separated from the programming model [4]. As a result, JXPL objects can easily be nested and evaluated recursively, which

enables JXPL scripts to be written dynamically while preserving the flow of execution.

The ability to nest objects and evaluate them recursively makes it possible to accurately depict dependencies between tasks in a workflow. This feature of the JXPL language is especially useful in a graphical workflow environment because workflows are inherently nested or recursive. In fact, Ptolemy II, the foundation of the GridNexus GUI, uses an XML modeling markup language named MoML to describe Ptolemy II models. However, MoML is a modeling language, not a scripting language [6].

4. JXPL Syntax

The main goal of this paper is to describe JXPL in terms that are specific enough that someone can use the language, or develop a tool to produce JXPL scripts. In this section, we provide a grammar for JXPL. Also, because JXPL scripts must be well-formed XML documents, we highlight the syntax rules for all well-formed XML documents. It is our intention that, by integrating these means of language description, we will provide a rich and useful reference for the JXPL language.

We begin our description of JXPL with the syntactic requirements shared by all well-formed XML documents. All XML documents must have a root element, and the root element for a JXPL script must be a single expression: either a JXPL Atom or a JXPL List. Tag names may be qualified with the JXPL namespace, which is **http://www.jxpl.org/script**. All elements must have closing tags, elements are case sensitive, and both tags must be the same case. Elements must be properly nested, and attribute values must be quoted [7].

Figure 2 presents the EBNF grammar for the JXPL language. We have omitted the rules for XS:STRING and XS:INTEGER, which are provided in the XML 1.0 specification [5]. We use the following conventions in the JXPL grammar: **x|y** means one of either **x** or **y**, **{x}** means zero or more occurrences, **[x]** means zero or one occurrence, terminals are underlined, and non-terminals are capitalized.

```
JXPLEXPR  → ATOM | LIST | ε
LIST      → <list> {JXPLEXPR} </list>
ATOM      → SYMBOL | STRING | INTEGER |
            RATIONAL | DECIMAL |
            PRIMITIVE
SYMBOL    → <symbol name=”XS:STRING”> </symbol>
PRIMITIVE → <primitive name=”XS:STRING”> {PROPERTY}
            </primitive>
PROPERTY  → <property name=”XS:STRING”value=”XS:STRING”>
            </property>
STRING    → <string value=”XS:STRING”></string>
INTEGER   → <integer value=”XS:INTEGER”></integer>
DECIMAL   → <decimal value=”XS:INTEGER[.XS:INTEGER]
            [scale=”XS:INTEGER” ] “>
            </decimal>
RATIONAL  → <rational numerator=”XS:INTEGER”
            denominator=”XS:INTEGER”>
            </rational>
```

Figure 2. Extended Backus-Naur Form (EBNF) for the JXPL Language

The primitive construct is used to describe a function call. XS:STRING type attributes may contain the legal characters of Unicode and ISO/IEC10646, line feeds, carriage returns, and tab characters. XS:INTEGER may contain an integer of arbitrary length [5]. Because the language is XML, any tag for which there is no nested information may be opened and closed in a single tag as per the rules of XML. For example, the tag **<integer value=”3”></integer>** may also be written as **<integer value=”3”/>**. For simplicity, we have omitted these extra rules from the grammar.

To demonstrate the grammar, Figure 3 shows a simple example of JXPL. This example uses the function named **Arithmetic**. The function is called with one **property** named **operation** which has the value **multiply** and two operands which are the rational numbers 7/5 and 3/11. The result of evaluating this script is:
<rational numerator="21" denominator="55"/>.

```
<list>
  <primitive name="Arithmetic">
    <property name="operation" value="multiply"/>
  </primitive>
  <rational denominator="5" numerator="7"/>
  <rational denominator="11" numerator="3"/>
</list>
```

(a) JXPL Example

```
(* 7/5 3/11)
```

(b) Equivalent in Lisp

Figure 3. Multiplication using the JXPL Arithmetic Function

4.1 JXPL Simple Types

JXPL simple types correspond to Lisp Atoms and include symbols, strings, integers, rationals, decimals and function names. However, in contrast to Lisp, atoms are not implicitly typed. Instead, the type is apparent in the name of the tag. This may limit the expressiveness of the language but specifying type reduces the complexity of the JXPL interpreter. Like Lisp, JXPL does not provide an explicit Boolean type. **False** is represented as an empty list **<list/>**, and everything else is considered to be **true**.

The JXPL symbol element is functionally similar to that of the Lisp symbol. In many other languages, identifiers, variable names, function names, and constants are viewed as separate elements. Instead, the symbol is a multipurpose element, which is an instrumental part of the powerful abstraction capabilities in both languages. For example, in JXPL a symbol can legally represent the number ten, the string value "ten", a function named "ten", or a variable named "ten".

JXPL includes three defined numeric types: integer, decimal and rational. The integer numeric type may be positive or negative but must be a numeric value without a

fractional component. The decimal numeric type may be positive or negative and may contain a decimal point. However, the decimal type in JXPL may also contain a scale property, which specifies the number of decimal places. If the scale property is not specified, the JXPL interpreter uses a default scale value of twenty. The rational numeric type requires both a numerator and denominator of integer types.

JXPL implements the numeric types using either the BigInteger or BigDecimal Java classes. BigDecimal provides support for immutable, arbitrary-precision, signed decimal numbers. A BigDecimal has an arbitrary precision integer unscaled value and a non-negative 32-bit integer scale, which represents the number of digits to the right of the decimal point [8]. BigInteger provides support for immutable, arbitrary precision integers and all operations behave as if integers were represented in two's complement notation [9].

4.2 JXPL Primitives

The JXPL primitive is analogous to a function. The primitive tag is used to call a function. The syntax of a function call with the primitive tag is:

```
<list>
  <primitive name=”function name”>
    properties
  </primitive>
    arguments
</list>
```

The function name is provided as an attribute to the primitive tag. A list contains the primitive and arguments. Similar to Lisp, the JXPL interpreter assumes that the first atom of a list is a function with its arguments following.

JXPL has many predefined functions. In this section we discuss the functions that are most relevant to grid computing: *Defun, WSClient, GSClient,* and *Prog*.

4.2.1. Defun. JXPL is designed to be extensible. *Defun* allows users to define their own functions. Figure 4 is an example of a user-defined function called *Myadd* that uses the JXPL *Defun* primitive. The *Myadd* function is a simple function that defines the operation x + 2. The syntax of *Defun* is:

```
<list>
  <primitive name =”Defun”/>
  <symbol value=”function name”/>
  <list> parameters </list>
  Function body
</list>
```

```
<list>
 <primitive name="Defun"/>
 <symbol name="Myadd"/>
 <list>
  <symbol name="x"/>
 </list>
 <list>
  <primitive name="Arithmetic">
  <property name="operation"
  value="add"/>
  </primitive>
  <symbol name="x"/>
  <integer value="2"/>
 </list>
</list>
```

(a) Defining Myadd

```
<list>
  <primitive name="Myadd"/>
  <integer value=”4”/>
</list>
```

(b) Calling Myadd

Figure 4. Defining and calling the function Myadd

4.2.2. Grid Primitives. Two of the most significant functions are the *WSClient* and the *GSClient*. These functions are designed to be generic clients for web and grid services. A detailed discussion of web and grid services is outside the scope of this paper. For more information on how these services operate, we refer the reader to [10]. Here we assume the reader has a basic understanding of these topics.

To use the Web Service Client function, the URL of the Web-Services Description Language (WSDL) is needed to generate a call to the appropriate method of the web service. The arguments to the WSClient are the WSDL, the name of the method of the service to be called, and any arguments to that method. Figure 5 shows the JXPL script that is produced for a simple web service called *MyMath.* This service has a method called “squared” which returns the square of its argument. The result returned from the script in Figure 5 is **<integer value=”4”/>**.

```
<list>
  <primitive name=".ws.WSClient"/>
  <string value=
        "http://beowulf.bear.uncw.edu:8080/axis//MyMath.jws"/>
  <string value="squared"/>
  <integer value="2"/>
</list>
```

Figure 5. JXPL script request to MyMath Web Service and result

The implementation and use of a generic grid service is more complicated. The GSClient needs the URL of the Factory service, the grid service instance name, the port type class name, and the stub class name. This information is provided to the GSClient via property tags. The method name and its arguments are provided in a list that follows. Also, both the service stub and service jar files need to be in the class path where the JXPL interpreter will execute. Figure 6 below shows the JXPL script for a grid service client configured to communicate with a grid service called *MathService*. In this example, we are contacting the Factory only, not a specific instance.

```
<list>
 <primitive name=".ws.GSClient">
  <property name="instanceName"/>
  <property name="factoryUrl" value=
   "http://beowulf.bear.uncw.edu:8080/ogsa/services/uncwCounter/
                                                    MathService"/>
  <property name="portType"
            value="uncwCounter.stubs.MathService.MathPortType"/>
  <property name="stubName" value="uncwCounter.stubs.
             MathService.bindings.MathServiceSOAPBindingStub"/>
 </primitive>
 <list>
  <string value="add"/>
  <integer value="1"/>
 </list>
</list>
```

Figure 6. Simple Grid Service Client

4.2.3. Prog. Since XML files must have a single root element, this restricts JXPL scripts to having a single workflow. However, we can have several workflows, which may or may not have a dependency relationship, contained within one script by using the Prog function. The Prog function accepts a list of lists and evaluates each sublist individually in the order in which they appear. The return value is the result of the last list evaluated.

In order to separate the GUI from the execution of a workflow, the JXPL interpreter can be run as a grid service, or it can be run remotely using Remote Method Invocation (RMI) and through a custom HTTP server. The Prog function has a property called **name**, for providing the URI of the interpreter that should execute its sublists. Every JXPL interpreter has a URI that indicates the protocol used to contact it. For example, a URI prefix such as "ogsa://..." indicates an interpreter that is running as a grid service, "rmi://..." indicates an interpreter that is contacted through RMI, and "http://..." indicates an interpreter this is available as a web service. If the processor is local, then the URI prefix may contain the word **local** or simply be omitted. The local processor inspects the URI of the Prog, and if the **name** property is absent, has no prefix, has a prefix of **local**, or has a URI that represents the current interpreter, then it evaluates the Prog sublists locally. Otherwise, the interpreter sends the Prog script to the remote interpreter for evaluation and waits for its response.

The Prog function also has a property called **fork**, which indicates that the current interpreter should create a new thread to execute the sublists. This is useful for allowing the current interpreter to return immediately while processing takes place in the background, either remotely or locally. Since it is possible to have multiple threads executing Prog's on various processors, a mechanism for synchronization is needed. This is accomplished through the property called **waitfor**. The **waitfor** property takes the URI name of another Prog whose termination must precede the evaluation of this Prog.

Figure 7 shows an example of using Prog's. The first Prog runs locally and evaluates the sub-Prog's. Two new threads are created on the two machines **bec** and **firedev** to evaluate each of the two sub-Prog's. The local Prog returns immediately with a result of **true**. The interpreter on **bec** waits until Prog2 completes before starting the evaluation of Prog1.

```
<list>
 <primitive name="Prog"/>
<list>
 <primitive name="Prog">
  <property name="name" value="{ogsa://bec.bear.uncw.edu:8080/ogsa/
                                          .../JxplInterpreter}prog1"/>
  <property name="fork" value="true"/>
  <property name="waitFor">
  <symbol name="{ogsa://firedev.bear.uncw.edu:8080/ogsa/
                                          .../JxplInterpreter}prog2"/>
  </property>
 </primitive>

Prog1 does some computation here after Prog2 completes
</list>

<list>
 <primitive name="Prog">
  <property name="name" value="{ogsa://firedev.bear.uncw.edu:8080/
                                    ogsa/.../JxplInterpreter}prog2"/>
  <property name="fork" value="true"/>
 </primitive>

Prog2 does some long computation here
</list>
</list>
```

Figure 7. Example of Prog Function

Another feature that is necessary once JXPL scripts are being executed in multiple threads and multiple processors is the ability to access remote data. This is accomplished by allowing symbols also to have URIs. If the URI of a symbol indicates that it is non-local, then the local interpreter makes a request for its value from the remote interpreter provided in the URI of the symbol. For example, if Prog2 in Figure 7 above sets a variable called **x**, then Prog1 could access **x** by:

```
<jxpl:symbol name=
      "{ogsa://bec.bear.uncw.edu:8080/ogsa/.../JxplInterpreter}/x"/>
```

5. Future Work and Conclusions

In this paper, we provided a specification of the JXPL scripting language. It is an XML-based scripting language inspired by LISP. JXPL is the language used to describe and execute workflows created using the GridNexus workflow editor. JXPL includes functions that serve as generic grid and web services, making it easy for novice users to create workflows that interact with these services. There are several nice features of JXPL:

1. The execution is separate from the graphical interface allowing for batch mode and parallel execution
2. JXPL is based on Lisp, making it elegantly simple and easily extensible
3. JXPL uses XML so that it can easily be transmitted via SOAP
4. Different parts of a workflow can be executed by different processors using different communication protocols (OGSA, RMI, or HTTP)

5. Because the JXPL interpreter is implemented in Java, it is easily ported to different platforms
6. JXPL can easily make use of available and idle processors

The first five features above lead up to the last one. Previously, making use of the free cycles of various computers owned by an organization was not an easy task. In particular, one would need to install special software such as the Globus toolkit or MPI and possibly an alternative operating system such as Linux. One could also develop a Java application that would allow a user to farm out work to idle processors. However, this is precisely what JXPL is. To make use of idle processors with JXPL, one simply downloads and runs a jar file of the JXPL interpreter. Using GridNexus, the user can specify that certain parts of the workflow be executed on the available machines. Although this requires the user to download the jar file, we are considering pushing the file to the available machines, making the setup even easier.

We are in the process of experimenting with this technology. We are currently using JXPL to run a primality prover based on a deterministic polynomial time algorithm by Agrawal, Kayal, and Saxena [11]. The key to this algorithm is that the work can be divided into independent tasks. Using 34 idle machines running Windows XP, we have been successful in proving the primality of a 25 digit number in 1 minute, a 50 digit number in 15 minutes, and an 80 digit number in 5 hours and 40 minutes. We are not yet ready to publish the results of this experimentation since we are still developing this technology and we have not yet established a baseline against which to compare the performance. However, we are confident that we will be able to demonstrate speedup since the primality proving algorithm uses independent tasks and we have several hundred desktop computers available for use.

Another application that we are in the process of developing is animation rendering, which is also computationally expensive but can be done with independent tasks. We hope to make it even easier to create new applications that can run in a distributed environment. We would like to encourage others to contribute to JXPL as well as to create new applications that will use JXPL.

6. Acknowledgements

This work is supported in part by the University of North Carolina Office of the President, UNC Wilmington's Information Technology Systems Division, and the National Science Foundation under Grant No. DBI0234520.

7. References

[1] J.L Brown and C.S. Ferner. GridNexus and JXPL: A Grid Services Workflow System. *globusWORLD* Boston, MA, February 7-11, 2005. http://www.globusworld.org

[2] R.St. Amant and M.R. Young, "Links: common lisp resources on the web," *Intelligence*, vol. 12, no. 3, pp. 21-23, 2001.

[3] K.H. Sinclair and D.A. Moon, "The philosophy of lisp," *Communications of the ACM*, vol. 34, no. 9, pp. 41-46, Sept. 1991.

[4] R.W. Sebesta, *Concepts of Programming Languages*, 4th ed. Reading, MA: Addison Wesley Longman, Inc., 1999, pp. 120-122, 569-572.

[5] F. Yergeau. (2004 February). Extensible Markup Language (XML) 1.0 (Third Edition). W3C Recommendation. Cambridge, MA. [Online]. Available: http://www.w3.org/TR/2004/REC-xml-20040204

[6] E.A. Lee and S. Neuendorffer, "MoML-A Modeling Markup Language in XML-Version 0.4," University of California at Berkeley, Tech. Memo. ERL/UCB M 00/12, Mar. 2000.

[7] J.E. Refsnes. (2004 October). XML Schema Tutorial. Refsnes Data. Norway. [Online]. Available: http://www.w3schools.com/schema/default.asp

[8] Sun Microsystems, Inc., BigDecimal (Java 2 Platform SE v1.4.2). [Online]. Available: http://java.sun.com/j2se/1.4.2/docs/api/java/math/BigDecimal.html

[9] Sun Microsystems, Inc., BigInteger (Java 2 Platform SE v1.4.2). [Online]. Available: http://java.sun.com/j2se/1.4.2/docs/api/java/math/BigInteger.html

[10] J. Joseph, M. Ernest, and C. Fellenstein, "Evolution of grid computing architecture and grid adoption models," IBM Systems Journal, vol. 43, no. 4, pp. 624-645, 2004.

[11] R. Ramachandran, "A prime solution," *Frontline: India's National Magazine*, vol. 19, no. 17, pp. 1-7, Aug. 2002.