The Encapsulation of Legacy Binaries Using an XML-Based Approach with Applications in Ocean Forecasting

by

Robert C. Chang

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2003

Author . . . .
Department of Electrical Engineering and Computer Science
May 21, 2003

Certified by . . . .
Nicholas M. Patrikalakis
Kawasaki Professor of Engineering
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Professor of Electrical Engineering and Computer Science
Chairman, Department Committee on Graduate Students

The Encapsulation of Legacy Binaries Using an XML-Based Approach with Applications in Ocean Forecasting

by

Robert C. Chang

Submitted to the Department of Electrical Engineering and Computer Science
on May 21, 2003, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis presents an XML-based approach for the encapsulation of legacy binaries. A method that utilizes XML documents to describe the various parameters and settings for the compilation and execution of an encapsulated binary is discussed. The binary is treated as a black-box component and the XML description for that binary contains relevant restrictions, such as input and output files and runtime parameters read in from the standard input stream.

The proposed XML schema design constrains the aforementioned XML descriptions of binaries. The usage parameters for the binaries are expressed by such XML documents. A prototype system is then able to take any of these schema-conforming XML descriptions and display the relevant user controls in a graphical user interface (GUI). Instead of editing obscure script files, the user can make changes to build-time and runtime parameters for a binary using the presented system interface. After validating the user inputs, the system generates the required script files automatically and proceeds to compile and/or execute the binary. The Primary Equation Model binary of the Harvard Ocean Prediction System (HOPS) was successfully encapsulated using the presented approach. The customization and control of the binary's compilation and execution through a GUI was achieved.

Thesis Supervisor: Nicholas M. Patrikalakis
Title: Kawasaki Professor of Engineering

Acknowledgments

This work was funded in part from NSF/ITR under grant EIA-0121263, and from the US Department of Commerce under grant NA86RG0074 (NOAA via MIT Sea Grant).

There are many people I would like to thank for helping me achieve my goals thus far and making this thesis a reality.

First of all, I would like to thank Professor Nicholas Patrikalakis for giving me this wonderful opportunity to be a part of the Design Laboratory and the Poseidon project. He provided a lot of guidance and suggestions for the research related to my M.Eng. thesis.

Dr. Constantinos Evangelinos also played a vital role in the completion of my thesis work during this past year. Extremely knowledgeable and understanding, he helped to inspire and polish many of the ideas that I've had this year regarding my project.

I would also like to thank my good friends for helping me get through my final year at MIT and making it enjoyable. Thanks Ed, Jen, and Chen, for always putting up with my antics and corny jokes. Thanks Amy and Rusan, for always looking out for me and showing me that there's much more to life than what we see before us.

Last, but not least, I would like to thank my family for always supporting everything that I do. I could not have gotten this far without you guys. Thanks Mom and Dad, for all the sacrifices you've had to make to get me here. This thesis is dedicated to my family.

For a better world...

Contents

List of Figures

Chapter 1

Introduction

The Design Laboratory of the MIT Department of Ocean Engineering is working on an NSF-funded project entitled, "Poseidon – Rapid Real-Time Interdisciplinary Ocean Forecasting: Adaptive Sampling and Adaptive Modeling in a Distributed Environment" and on a related US Department of Commerce project (funded by NOAA via MIT Sea Grant) entitled "Poseidon: A Coastal Zone Management System via the World Wide Web" [16]. The overall goal of the project is to contribute to the development of modern interdisciplinary ocean science by combining advanced information technologies with ocean sciences to enable the efficient real-time forecasting of dynamic physical and biological events in the ocean and further advancements in oceanic sciences.

The work discussed in this thesis was developed within the context of the Poseidon project, as part of a network of distributed heterogeneous software resources and data. The first of two main components in the thesis deals with the design of an appropriate XML schema to constrain the XML description files of the legacy binaries used in the project. The other major component is the implementation of a prototype system that processes these XML descriptions and generates relevant controls for the user to build and execute the binaries graphically. The implemented system is also capable of being deployed as a Web front-end.

1.1 Poseidon

Effective ocean forecasting is essential for successful human operations in the ocean. Recent developments in the availability of high-performance computing and networking infrastructure have enabled the construction of distributed computing systems that address computationally intensive problems in interdisciplinary oceanographic research. By using effective interchange mechanisms to enable free collaboration and interdisciplinary ocean research activities, researchers will be able to speed up their computing for better simulations and allow more time for research. Therefore, there is a great need for a modern distributed computing and networking infrastructure for scientific research [15].

Poseidon is a distributed computing infrastructure that brings together advanced modeling, observations tools, and field and parameter estimation methods for oceanographic research. The Poseidon project aims to enable efficient interdisciplinary ocean forecasting by creating a dynamic data-driven forecast using an operational distributed computing framework. It provides seamless access, analysis, and visualization of experimental forecast data through a user-friendly Web interface that conceals the complex framework of hardware and software resources.

1.2 Harvard Ocean Prediction System (HOPS)

The Poseidon project utilizes the Harvard Ocean Prediction System (HOPS) [18] as its underlying advanced interdisciplinary forecast system. HOPS is a portable and generic system for interdisciplinary nowcasting and forecasting through simulations of the ocean. It provides a framework for obtaining, processing, and assimilating data in a dynamic forecast model capable of generating forecasts with 3D fields and error estimates. The HOPS system has been applied successfully to several diverse coastal and shelf regions [19], and analyses have indicated that accurate real-time operational forecast capabilities were achieved. However, as powerful as HOPS may be, the software used in the system is still based on legacy Fortran binaries that do not fit well within the modern distributed computing model of operation.

1.3 Motivations

One of the initial problems encountered during the design process of the Poseidon system dealt with the fact that HOPS (as well as other ocean modeling systems, such as ROMS [7]) is a *legacy* program, like many scientific applications. The term “legacy” refers to software not developed using the more recent programming languages (i.e. Java and C++) or lacking a graphical interface, and should not be misunderstood to imply obsolete code in this context. A legacy program could still have an active development community and incorporate contemporary software algorithms and techniques. Legacy programs oftentimes consist of compiled binaries that expect a standard input (*stdin*) stream, maybe some command line options, and a set of input and output files. In such setups, a workflow of binaries is executed either interactively (a common approach), or hard-coded in scripts. While such an approach, which originated from the days when graphical user interfaces (GUIs) were not available, is efficient for an experienced and skilled user, it is cumbersome and error-prone, and involves a steep learning curve. Furthermore, this approach does not adapt readily for the remote use of programs over the Internet.

Various methods for handling this issue in the framework of the Poseidon distributed computing architecture were examined during the design, and the robustness of the system to allow for future adaptation to non-HOPS components was also taken into consideration. The final decision was to keep working with the Fortran binaries of HOPS and encapsulate their functionality and requirements using the eXtensible Markup Language(XML) [28]. The goal is to create a computer-readable manual for the program binaries. XML is a standardized format that easily allows for self-describing files containing any data. A GUI is generated according to the data within the XML description files, with additional capabilities to check for the correctness of program parameters and drive execution in a transparent manner.

1.3.1 Wrapping Legacy Programs

The software components used in Poseidon are written in legacy Fortran 77 code, as such they do not adapt well within the modern distributed computing model. For instance, there is a lack of support for dynamic memory allocation and object-oriented programming, both commonly supported by modern programming languages.

Legacy is an unavoidable effect of technological advancement. The aging software becomes more and more difficult to integrate with newer systems as time passes. A key limitation of legacy software is platform dependency. The program user interface is often confined to command line inputs and outputs in a console window, and the software is limited to specific machines and operating systems.

There are several options to update legacy software for use within the modern distributed model of computing. One such option is to migrate the code and rewrite the entire application in a modern language, such as Java. This option is the cleanest approach, but can be very risky and consumes a lot of valuable time and resources better spent elsewhere. It is impractical or even unfeasible to covert existing procedural programs to object-oriented components [24].

A second option is to program with frameworks, such as the widely used Java Native Interface (JNI) [13], in order to free the legacy software from the constraints imposed by its language. JNI is Java's native programming interface that allows applications and libraries written in other languages to work with Java. In the JNI framework, the legacy code is considered as native methods. JNI enables the native methods to take advantage of the Java programming language – these native methods are allowed to use Java objects and call Java methods. In a way, JNI serves as glue between legacy software and the Java programming language.

A third option is to encapsulate the legacy software with an XML interface. XML can be used to describe data through the use of custom-defined tags – eliminating the need to conform to a specific programming structure and offering the prospect of integrating legacy software with new technology and infrastructures. The work for this thesis takes the last approach and encapsulates the Fortran 77 binaries of HOPS components using XML.

The work presented in this thesis includes the design of an XML schema used to constrain the syntax and structure of the XML description files encapsulating the HOPS legacy binaries. Such XML description files specify the usage of program binaries and allow for their machine-controlled, automated manipulation. An implementation of a Web-based front-end is used to parse the XML descriptions and generate relevant user controls. The front-end is able to present the appropriate metadata to the user and validate the user input against possible constraints to ensure correct functionality by the HOPS binaries. As a result, the user is no longer limited to using the program console for the control of the HOPS binaries.

1.4 Related Work

There has been some related work done to date, in the development of techniques and tools for the encapsulation of legacy software. However, none of the various works focused on wrapping these legacy software components at the binary level to allow the users to modify the component runtime parameter values. Sneed [23] discusses techniques for encapsulating legacy COBOL programs with an XML interface. He divides these programs into three categories — online programs, batch programs, and subprograms. Based on the program type, different wrapping strategies and tools are utilized. He intends for the solution to promote communication between people within the mainframe COBOL community. Wrapping the COBOL programs will allow the COBOL community to preserve their state-of-the-art while enabling others outside their community to benefit from the COBOL programs as well. A notable limitation to the wrapping technique described in this paper is its necessity to alter the legacy components within an architecture. The component alteration is done in order to adapt the components for reading and writing XML interfaces.

A series of papers [17, 22, 25, 26] illustrate the software architecture of a problem-solving environment (PSE) used for the construction of scientific applications from software components. These papers refer to the PSE as an integrated computing environment for composing, compiling, and running applications using an XML-based component model. Users visually construct domain-specific applications using a Java/CORBA-based problem solving environment for scientific simulations and computations. Software components, wrapped as CORBA [3] objects, are pieced together – independent of location, programming language, and platform. Each encapsulated component has its interface and constraints defined in XML.

Walker, Li, and Rana demonstrate in [25] the wrapping of an MPI-based molecular dynamic simulation program, written in C originally, into Java/CORBA objects with XML interface. The fundamental infrastructure for component interface definitions was Java IDL, a CORBA-compliant IDL. A client would invoke the wrapper for legacy code through IDL, and submit input data for simulation without knowing the wrapper location and the specific implementation of the simulation software. A key benefit to this approach is that users can supply simulation data to the molecular dynamic simulation program for simulation results without ever downloading the program.

There are some main differences between the approach taken in the work for this thesis and the approaches shown above. The aim is to encapsulate software components without having to adapt the components for XML. The emphasis is on treating the components as black-box objects, such that no changes to the binaries need to be made in order to interface with XML. Using XML interfaces, the eventual goal is to be able to piece the encapsulated components together as workflows, much like the PSE presented by Walker, et al. However, this implementation would not be tied to the CORBA infrastructure like the described PSEs. CORBA provides a framework for inter-operating objects that are implemented in different languages. While powerful, its mechanism requires the knowledge of implementation details of every shared object in order for clients to be able to access the methods for each component. By contrast, my work treats all the software components as black-box objects and allows for user modifications to the runtime parameters. By encapsulating at the binary level, there is no need to break the existing Fortran code into separate callable procedures as required for CORBA wrappers.

1.5 Outline of Chapters

The organization of this thesis is as follows:

- Chapter 2 provides an introduction to the various technologies used in the work associated with this thesis.
- Chapter 3 discusses the XML schema design developed and describes the resulting XML description files that conform to the schema design.
- Chapter 4 describes the implementation of the prototype system, with its graphical interface and features.
- In Chapter 5, the preliminary results from the adaptation of the schema/system to HOPS programs are given.
- Chapter 6 gives the conclusion and outlines some of the limitations of the current setup. It also provides suggestions for future research and implementation goals.

Chapter 2

Background

The following sections describe the technology and underlying concepts relevant to the thesis work and its motivations.

2.1 XML and Related Technologies

2.1.1 XML

XML stands for eXtensible Markup Language. It is a metalanguage – a language used to define new markup languages. The present work uses XML for its inherent ability to describe data and still remain platform-independent. XML is a standard for data representation and exchange on the Internet. Unlike HTML, which was designed to display data, XML was designed to describe data [27]. HTML tags define how a document should be displayed, while XML tags relate to the meaning of the enclosed text. XML allows developers to design custom tags and data structures for application-specific situations, and provides standardized data formatting used in cross-platform exchange of information. This standardization is made possible through the use of Document Type Definition (DTD) and XML schema language, both discussed in later sections.

Two key issues to consider for any XML document are the well-formedness and validity of the document. All legitimate XML documents must be well-formed. This

```
<list>
   <item>
      <name>Chair</name>
      <price>59.99</price>
   </item>
   <item>
      <name>Table</name>
      <price>129.99</price>
   </item>
</list>
```

Figure 2-1: Sample XML Document

means that no out-of-order nesting of tags exist and every open tag is closed. Only well-formed documents can be handled correctly by XML parsers. Figure 2-1 shows a sample well-formed XML document.

In addition, a document is not required to have validity; although it should have validity to ensure successful machine processing. An XML document can conform to a DTD/schema, which defines the grammar and tag set for a specific XML formatting. Since XML tags are not predefined, XML documents require Document Type Definition (DTD) or XML schema to describe the legal structure, constraints, and contents of XML documents. Conformity to a DTD/schema ensures that an XML document can be understood and processed by different machines.

2.1.2 DTD

DTD (Document Type Definition) defines the legal structure of an XML document. It establishes a set of constraints specifying the valid tags for a document and providing rules for how the document should be constructed [20]. This is particularly important for data transfer between applications, as there must be a pre-specified formatting scheme and syntax for various computer systems to interface with each other. DTD allows each XML document to be validated and processed by any machine that has access the document's DTD.

DTD has some rather serious limitations. DTD documents have no hierarchy

```
<!ELEMENT list (item+)>
<!ELEMENT item (name, price)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT price (#PCDATA)>
```

Figure 2-2: DTD for XML Shown in Figure 2-1

and the resulting structure is very flat (evident in the sample DTD provided as Figure 2-2). In addition, DTD has difficulty handling namespace conflicts, and the datatyping offered is very limited. DTD treats the contents of all tags contents as characters. Furthermore, DTD has no means for specifying allowed relationships between XML documents. Even though it is excellent for validating the structure of an XML document, DTD uses a different syntax than XML. Figure 2-2 provides a sample DTD for the XML example presented previously.

2.1.3 XML Schema Language

The XML schema language is an XML-based alternative to DTD [20]. The XML schema language was designed to replace DTDs by offering an XML-centric method of constraining XML documents. A DTD does not share the hierarchical structure of XML, which has caused much confusion; the schema language resolves this issue by using XML itself to define XML documents. XML schema documents are actually XML documents that are both well-formed and valid. This allows parsers and other XML applications to handle schema documents in a fashion similar to ordinary XML documents, instead of using special techniques required for handling DTD documents. Figure 2-3 shows the XML schema for the XML document from Figure 2-1.

An important advantage of XML schema is its rich support for datatypes commonly used by ordinary programming languages like Java. This makes it possible to provide document validity and work with data from various sources and platforms. Whereas DTD treats all data as characters, the W3C XML schema specification [29] has predefined datatypes and also allows for the definition of new ones. This powerful datatyping capability simplifies the processes of creating and validating XML

```
<element name=``list''>
   <element name=``item''>
      <complexType>
         <element name=``name'' type=``string'' />
         <element name=``price'' type=``double'' />
      </complexType>
   </element>
</element>
```

Figure 2-3: Schema for XML Shown in Figure 2-1

documents, so schemas offer a notable improvement over DTDs.

The schema language supports inheritance and the division of a schema into various components. Thus, new schemas can be created from existing schemas and predefined components can be referred to when writing schemas. These features increase the efficiency of software reuse and improve the XML software development process.

2.2 XML Parsing Technologies

2.2.1 XML Parsers

The parsing of XML documents is a pivotal aspect of XML programming – the data contained within the XML document becomes available to other applications only after the document is parsed. Therefore it is important to choose a suitable XML parser based on performance and functionality requirements. Since the programming work for the thesis is done in Java, a Java-based XML parser seems appropriate.

Two criteria used in parser selection for the project are the parser's conformity to XML and XML schema specifications and the ability of the parser to validate XML documents against DTD and XML schema. Since XML schema has a crucial role in the project, the chosen parser should have good support of the W3C XML Schema Recommendation [29].

The major Java-based schema-validating parsers are from the following organizations:

- Apache: http://xml.apache.org/xerces2-j
- IBM: http://www.alphaworks.ibm.com/tech/xml4j
- Microsoft: http://www.microsoft.com/xml
- Sun: http://java.sun.com/products/xml

Of course, there are several Java-based XML parsers smaller in size and with faster performance, such as Piccolo[1] and XP[2]. However, since these parsers either don't provide validation or don't offer support for XML Schema, only the major parsers with the right features were considered.

Microsoft's parser was not given much consideration since their implementation does not conform to W3C's XML specification. The results are mixed for various parser performance tests found on the Internet. According to a test conducted by DeveloperLife[3], IBM's XML Parser for Java (XML4J) outperformed Sun's Project X parser; a separate test conducted by DevX[4] showed the opposite result.

The Apache Xerces-J and IBM (XML4J) parsers both implement most of the W3C XML Schema specification [29] and offer support for the Java API for XML Processing (JAXP 1.1). These two parsers are comparable since they both stem from IBM research development and offer similar support for specifications and functionality. Their similarities are due to IBM being a major contributor to Apache's Xerces-J code base, which forms the basis for the XML4J 4.0.1 parser. In 1999 IBM released the source code to the community that was building the technology by donating the code to the Apache Software Foundation.

In the end, the Apache Xerces-J parser was chosen for this project since all the major parsers were comparable in capabilities, and the performance results found were mixed and inconclusive. The Xerces parser does not have corporate influences like the other parsers, and seems to be a popular and reliable parser offering support

[1]http://piccolo.sourceforge.net
[2]http://www.jclark.com/xml/xp
[3]http://developerlife.com/parsertest2/performance.html
[4]http://www.devx.com/xml/argicles/pm020101/pm020101-4.asp

for W3C's specifications. It is also one of the most widely contributed-to parsers available.

2.2.2 SAX

The Simple API for XML (SAX) [21] provides an event-based framework for the XML parser to use when parsing XML data, which is to go through the XML document and break down the enclosed data into usable chunks. One issue to clarify at this time is that SAX is not an XML parser. It simply provides a framework for parsers to use, and defines the events to monitor during the XML parsing process. The SAX APIs only provide the means to parse XML documents.

SAX is an event-driven model, which means that events are defined by SAX to occur during the parsing process. The programmer provides the callback methods to be invoked by the XML parser as it processes the XML data. This allows the handling of the various situations that can occur during parsing. For example, event handlers can be defined to output the time when the parser encounters the beginning and end of the document. The difference between these two times can then be used to calculate the total time required by the local machine to process the given XML document. Some possible parameters used to generate SAX events include XML elements, attributes, and comments.

SAX is a popular protocol since it is the fastest and the least memory-intensive method of dealing with XML documents. On the other hand, SAX also requires much more programming than other common APIs, such as DOM (described in the next section). Another disadvantage to using SAX is its sequential nature. It gives users linear access to the contents of XML documents and cannot back up to an earlier part of the document. There is no manipulation of the parsed data.

2.2.3 DOM

DOM stands for Document Object Model [4] and is designed to handle the manipulation of the parsed data (the shortcoming of SAX). DOM represents a parsed document as a tree structure and adapts readily to most programming languages, since the traversal and manipulation of tree structures can be accomplished easily by most programming languages.

DOM reads the entire XML document into memory, to provide for quick access to any part of the document structure. The data is stored as tree nodes, where each node contains one data component from the parsed XML document, and the resulting structure of the tree matches that of the original document. Since DOM reads entire documents into memory, it can be quite a burden on system resources. Larger and more complex XML documents can potentially cause significant degradations in the performance of the application/system.

2.2.4 JDOM

JDOM (Java Document Object Model) [12] is a new technology that enables Java developers to read, change, and write XML data much more easily than ever before. All previous programming libraries and APIs (application programming interfaces) designed to interact with XML were intended to work with multiple languages, which causes inefficiencies for Java programmers; whereas JDOM uses the power of the Java language to make interactions with XML simpler and faster.

JDOM is an open source API that provides an alternative to the standard APIs packaged with the Apache Xerces-J parser, such as the SAX and DOM APIs discussed in the previous subsections. Unlike DOM and SAX, JDOM uses standard Java programming idioms and takes advantage of Java language features. JDOM builds a tree structure from the XML document being parsed, much like DOM.

For the purpose of this thesis, the JDOM API was chosen for its more user-friendly and intuitive methods. The Web-based front-end will use the Apache Xerces-J parser to parse the XML descriptions of the legacy Fortran binaries. The system then uses the JDOM Beta 8 release[5] to build document models for application processing and manipulation of the parsed data. The JDOM API is used to interact with the XML being processed. These interactions range from the initial construction of the tree

[5]Most current as of August 2002

structure representing the XML document, to the changes to the elements as the user updates variable values using the front-end GUI.

JDOM is the prime candidate for handling XML documents in the work presented by this thesis, since all programming is done in Java. JDOM has close ties with Java and was written to be more intuitive to programmers than the DOM API. It encompasses the best features of the SAX and DOM APIs. It has the fast processing time and small memory footprint of SAX, while still being able to parse XML documents into tree structures, like DOM, for random access. Therefore, there isn't as much of a concern for performance degradation from handling larger, more complex documents using JDOM, and the framework is no longer limited by sequential access.

2.3 Java

All of the programming for this thesis is in the Java programming language. Even though Java is a relatively new programming language, it has become much more stable and reliable in recent years. The primary advantages of Java are its portability and networking capabilities, which can provide advanced features for the developed software to run over networks such as the World Wide Web.

With its first official release by Sun Microsystems back in early 1996 [8], Java provided programmers with a syntax similar to that of C++. Java is also fully object-oriented – everything in Java is an object (except for basic types like integers and Booleans). This gives it many advantages for dealing with complex projects.

The main reason for using Java in this thesis is its platform independence. This allows for the developed code to be run on any machine equipped with a Java interpreter. Java was designed so that the compiler generates architecture-neutral bytecode that is executable on many processors/platforms. The bytecode instructions have nothing to do with particular computer architectures, and can be interpreted on any machine and easily translated into native machine code. As long as an interpreter has been ported to a particular platform, Java bytecodes can be executed directly on that platform. This portability allows users to download compiled Java bytecode across the Internet and use local Java interpreters to execute the bytecode. Although there is a definite disadvantage with the slower performance from using bytecode compilers than from using a true native code compiler, users are not tied down to specific platforms for code execution.

2.3.1 Java Applets

Through applets, Java provides a mechanism for programs to work within webpages. To run a Java program within a webpage, the program must be converted to an applet first, since Web browsers cannot handle normal Java programs. Java applets are embedded in HTML pages, much like the way images and tables are embedded.

To use an applet, the Web browser must also be Java-enabled so that it can interpret the bytecode within applets. Sun Microsystems developed the Java Plug-in, which makes the newest Java runtime environment available to browsers so that users won't have to worry about a browser's default support for Java. Once the user installs the Java Plug-in for the Web browser, the browser is capable of interpreting Java applets within Web pages. Java applets don't have to be launched from the command-line like Java normal programs, and are much more convenient to use for the average user.

The prototype program developed for this thesis can run as either a standalone application or a Web applet. The Java code is compressed into a single file, which can be embedded in a Web page as an applet, or the user can choose to download the file to run it as a standalone application at the command-line prompt.

2.3.2 Security Issues with Java Applets

Since a Java-enabled browser allows Java code to be embedded within a Web page, downloaded across the Internet, and run on a local machine, security is a paramount concern. Users can easily download applets – exposing Java users to a significant number of risks. For example, when a user loads an unknown page into the browser, the page could contain a malicious applet that is automatically loaded and executed by the browser. The designers of Java were well aware of these risks associated with executable content, and therefore designed Java with security concerns in mind.

Java features a sandbox security model. This sandbox model confines Java applets, potentially malicious (in general), to a strictly defined environment where they cannot affect other system resources. Standalone Java applications are deemed to be trusted and enjoy unlimited access to all system resources, while applets are untrusted by default. This is because applications are downloaded with consent by the user, while applets can be downloaded even without the user's knowledge. The primary intent of the designers is to prevent untrusted applets from accessing and changing files on the local file system. There is also a need to prevent applets from using network connections to circumvent file protections or to act as malicious network agents.

By default, untrusted applets are prevented from reading and writing files on the user's local file system, and cannot make network connections except to the originating host of the applet. Furthermore, applets are unable to load libraries or start other programs on the user's local machine. With such restrictions, the prototype system developed for this thesis would be rendered useless and would not be able to perform most of its required tasks when run as an applet. The following is a sample list of the program tasks requiring access to system resources:

- Opening local XML files.
- Opening Web-based XML files.
- Saving script files locally.
- Running script files locally.

In order for browsers to trust an applet, the applet must be signed. The end user can then use a public key certificate sent with the applet to authenticate the signature. The Java 2 Software Development Kit (SDK) [9] provides several tools for dealing with security. The two tools of concern are *keytool* and *jarsigner* [10]. After using *keytool* to generate a public/private key pair, and signing the applet with the private key using *jarsigner*, users of the applet will see a security warning window pop up before the applet loads in the browser. This window will let the users view the certificate and prompt the user to grant or deny the necessary permissions for the applet to run properly.

Keytool

Keytool is used for the management of the *keystore* and certificates. The *keystore* is a repository that stores all keys and certificates for the system. The individual entries are accessible by unique aliases. *Keytool* creates public/private key pairs, issues certificate requests, designates public keys as being trusted, and handles X.509 certificates.

Keytool lets users specify key-pair generation and the signature algorithm used. The following command is used to generate a key using the RSA algorithm. The resulting key will be valid for 1000 days and will be accessible through the alias *robchang* in the keystore.

```
C:\java>keytool -genkey -alias robchang -keyalg rsa -validity 1000
Enter keystore password:
What is your first and last name?
  [Unknown]:  Robert Chang
What is the name of your organizational unit?
  [Unknown]:  Department of Ocean Engineering Design Lab
What is the name of your organization?
  [Unknown]:  Massachusetts Institute of Technology
What is the name of your City or Locality?
  [Unknown]:  Cambridge
What is the name of your State or Province?
  [Unknown]:  MA
What is the two-letter country code for this unit?
  [Unknown]:  US
Is CN=Robert Chang, OU=Department of Ocean Engineering Design Lab,
O=Massachusetts Institute of Technology, L=Cambridge, ST=MA, C=US
correct?
  [no]:  yes

Enter key password for <robchang>
        (RETURN if same as keystore password):
```

The generated key can be exported as a certificate and viewed with the following commands:

```
C:\java>keytool -export -alias robchang -file robchang.crt
Enter keystore password:
```

```
Certificate stored in file <robchang.crt>

C:\java>keytool -v -printcert -file robchang.crt
Owner: CN=Robert Chang, OU=Department of Ocean Engineering Design Lab,
O=Massachusetts Institute of Technology, L=Cambridge, ST=MA, C=US
Issuer: CN=Robert Chang, OU=Department of Ocean Engineering Design Lab,
O=Massachusetts Institute of Technology, L=Cambridge, ST=MA, C=US
Serial number: 3ea5bbab
Valid from: Tue Apr 22 18:01:15 EDT 2003 until: Mon Jan 16 17:01:15 EST 2006
Certificate fingerprints:
         MD5:  FE:30:B6:EB:50:0E:75:9F:41:2B:8F:DF:5E:F2:D7:73
         SHA1: 23:26:A6:8B:1F:6E:33:8A:DB:89:77:80:E3:51:FC:F4:AE:20:8B:E6
```

Jarsigner

The *jarsigner* tool accesses the keystore to locate the private key and its associated certificate to use for signing a *.jar* file. Only users who know the passwords to the keystore will be able to access a key in the keystore and use it to sign a *.jar* file. This is because passwords protect access to the keystore and its private keys. The following command is used to sign the *.jar* file for the applet with the *robchang* key created in the previous section.

```
C:\java>jarsigner prototype.jar robchang
Enter Passphrase for keystore:
```

Password Issue with Keytool and Jarsigner

There is an implementation oversight with the keytool and jarsigner tools in the Java SDK. Even though these tools provide extra security measures for Java development, the passwords entered by the user appear as plain text on the screen. The password typed into the command prompt is left unobscured on the screen. Programs commonly leave out the password characters or replace the characters with asterisks (*) to decrease the chance of the password being stolen. In the case of keytool and jarsigner, the typed passwords are shown in plain sight. The passwords entered in the examples above have been omitted.

A simple solution is to have the user clear the screen immediately after using the tools. This is extremely inconvenient and leaves the passwords exposed if the user ever forgets to clear the screen. There is another possible solution to this problem.

There exist programs that handle the security management tools of the Java SDK graphically. The user performs all the above tasks through a GUI, and these GUI-based programs usually make some attempt to obscure the entered password.

Chapter 3

XML Schema Design

3.1 Requirements and Overview

The XML interface to the encapsulated HOPS programs should be self-contained and should not require any modification to the programs. By providing a detailed XML description for each program, the program can be treated as a black-box component. The prototype system developed as part of this thesis should then be able to parse in the XML descriptions, and from their contents, determine the specifics on how to properly set up and run the program with the appropriate parameters.

Several key concerns have to be addressed and supported in the XML schema design. The resulting XML documents that conform to this schema should provide as much relevant information to the user as possible, so that the user can make well-informed decisions while customizing various build-time and runtime parameters. There should be a set of default parameter values so that manual entry of values for each program execution can be avoided, especially since there can be hundreds of parameters. It does not make any sense to require the user to enter in all the values every time the user runs the program with the prototype system.

The XML descriptions conforming to the proposed schema should also be capable of specifying the contents of the program makefile and its parameters, in order to allow the user to recompile the program binary as needed. The *makefile* is a file that tells the *make* utility how to compile and link a program. The makefile contains dependency information and variable/macro definitions, as well as standard shell-based commands. During the compilation process, the make utility compiles all the source files, and the resulting object files corresponding to the source files are linked together to produce the new executable binary. How a program compiles depends on the system architecture, preprocessor macros and definitions, and specific input values.

The resulting compiled binary could have input/output files and other parameters required during execution. Since runtime parameters should be checked for legality after any user modification, the schema design must support datatypes and additional constraints on the parameter values. Instead of treating all parameter values as plain strings, the introduction of datatypes and constraints helps to ensure the correctness of program parameters. This is to facilitate the compilation and execution of the program and ensure that all input parameters are acceptable. Each parameter value can be validated against its constraints before proceeding.

The proposed XML schema design consists of four major components. The first component corresponds to the top-level description of the encapsulated program. The program description contains the basic information about the program, its makefile, and the various compiled binaries available. The next component deals with the program makefile. This is useful for the automatic generation of a customized, platform-specific makefile used to compile the program. A third schema component constrains the XML descriptions of the compiled program binaries. This schema supports the description of a binary's input and output files. It is also capable of handling command-line arguments and other runtime parameter sources. The last part of the schema handles the descriptions of parameter files used during program compilation and execution. The parameter files contain build-time and runtime input parameters, and consist of customizable variable values for the program. The following sections will fully describe the schema design and discuss them in detail.

3.2 Program Schema

The program schema is the top-level schema for the encapsulated program. The program schema (*program.xsd*) is included in Appendix A. An XML file conforming to the program schema describes the fundamental aspects of an encapsulated program. The root *program* element contains seven major components. These component elements provide the key information concerning how the program is compiled and executed. The element descriptions are listed below:

name: The name of the program.

info: Information about the program.

architecture: A list of available platforms for the program and the selected platform.

shell: A list of available shells for running scripts, and the selected shell.

makefile: Information about the makefile and its path.

binaries: Information about the various compiled binaries for different platforms and their respective paths.

constraints: List of files used during the compile process.

Figure 3-1 shows the hierarchy of the program schema. There can be any number of the elements that denoted with a * symbol. The other elements can only appear once in an XML description file.

The *name* and *info* elements are standard for many components in the program schema, and the other schemas as well. These elements are useful for the human users, and helps them better understand the contents of the XML description files. The *architecture* element holds of a list of architectures available to the program and the architecture selected. The extra component for the selected architecture is useful for defining a default architecture for the program. The *architecture* element consists of the *choice* and *selected* elements. The *choices* element contains the list

```
-program
    -name
    -info
    -architecture
        -choices
            -choice*
        -selected
    -shell
        -choices
            -choice*
        -selected
    -makefile
        -info
        -type
        -path
        -xmlDesc
    -binaries
        -binary*
            -info
            -arch
            -path
            -xmlDesc
    -constraints
        -constraint*
            -name
            -info
            -path
            -xmlDesc
```

Figure 3-1: Hierarchy of Program Schema

of architectures, with each architecture embodied by a *choice* element. The *selected* element contains the selected architecture.

The elements for architecture choices and the selected architecture are all *archTypes*-type elements. The *archTypes* type is based on the string type. The allowed values for architecture are defined in the schema to be "alpha," "cray," "iris," "rs6000," "sun3," "sun4," "sun5," and "linux." These are the platforms currently supported by HOPS. Other architectures can certainly be included by expanding the definition of the program schema to allow for them. Better still, a standardized namespace could be adopted for the naming of the different architectures. This would help to prevent different naming conventions from being used for the same architecture.

The structure of the *shell* element is analogous to that of *architecture*. In order to compile or execute the program binary, the prototype system creates shell scripts containing commands to run with certain shell programs. There is a *choices* element with the list of available shells for scripts, and a *selected* element for the selected shell. These two elements are *shellTypes*-type, which is defined to contain one of the following values: "sh," "bash," "csh," "tcsh," and "ksh." These represent the common shells such as Bourne shell, C shell, and Korn shell.

The next major element is the *makefile*. This element consists of four parts: *info*, *type*, *path*, and *xmlDesc*. The *type* element denotes the type of make tool used on the makefile to compile the program. Some common make tools include BSD make and GNU make. The *path* element holds the path of the program makefile. The *xmlDesc* element contains the path of the XML description file of the makefile, if one is available. The makefile can have its own XML description file, which would conform to the makefile schema proposed in Section 3.3. The prototype system discussed in the next chapter can parse and understand XML description files conforming to this schema design. The system produces a graphical user interface (GUI) for the user to customize, and then it automatically generates a makefile based on the information provided in the XML description of the makefile and any additional user modifications. If the makefile does not have an XML description file, then the *xmlDesc* element is left empty.

The *binaries* element consists of a series of *binary* elements, each corresponding to a binary compiled for a specific architecture. Each *binary* element has four main components: *info*, *arch*, *path*, and *xmlDesc*. The *arch* element denotes the architecture for the binary. The *path* element shows the path for the program binary. The *xmlDesc* element holds the path for the binary's XML description file. Again, this element is left empty if the binary does not have a description file. Before moving on, there is one point to clarify here. When people talk about "running" a program, this is referring to executing a program's binary. A "program" can be thought of as the source code, which cannot be used unless the code is compiled into something executable. The program binary refers to this executable result from compiling the source code.

The last element is *constraints* – a list of parameter files used during the compiling of the program (e.g. include files containing constants, etc.). Each file is described by the *constraint* element, which includes four sub-elements. These elements are the *name*, *info*, *path*, and *xmlDesc*. The purpose of these elements has already been explained previously. The XML description of such parameter files must correspond to the parameter schema presented in Section 3.5.

3.3 Makefile Schema

The file for the makefile schema is included in Appendix B as *makefile.xsd*. The makefile schema constrains XML description files for program makefiles. A makefile is used to define the compilation process for software projects. Within a makefile are variable definitions and dependencies used for compiling the program source code and possible dependencies between these variables. The makefile also specifies the locations of the source files required and sometimes contains architecture-dependent portions that may or may not be used during program compilation.

The first two parameters of the top-level *makefile* element are the *path* and *info* elements. The *path* element holds the path of the program makefile on the local system. A *makefile* has a number of sections, described by *section* elements. The contents of each section could be from a file fragment, or could be a list of makefile preprocessor macros and definitions. Each *section* element has an *info* element and one of three content elements to choose from (*includeFile*, *includeFileChoice*, or *preproc-objects*), depending on the section's contents.

Figure 3-2 provides the makefile schema hierarchy. The elements that have a * symbol next to them can have any number of occurrences. The # symbol is a reminder that the *section* element can only include an *info* element and one of three element choices (*includeFileChoice*, *includeFile*, *preproc-objects*).

The *includeFile* element is used for including files as part of the makefile. The *includeFile* element is treated as a string and contains the path of the file to be

```
-makefile
    -path
    -info
    -section#
        -info
        -includeFileChoice#
            -choice*
                -architecture
                -includeFile
        -includeFile#
        -preproc-objects#
            -startText
            -endText
            -separator
            -preproc-obj*
                -name
                -info
                -header
                -value
                -use
                -requires
                    -item*
    -conflicts
        -conflict*
            -item*
```

Figure 3-2: Hierarchy of Makefile Schema

included. For example, by defining a makefile section with an *includeFile* element for the file *segmentA.txt*, the contents of *segmentA.txt* is included as part of the makefile. This is useful for the common parts shared by a program's makefiles for different architectures.

The *includeFileChoice* element is available for including architecture-specific portions of the makefile from various files. This is similar to the *includeFile* element in that a file is included in the makefile. However, the file that is included is dependent on the selected architecture for the program. The compilation process for a program could require a separate makefile for each of the various architectures it supports. The common parts of these makefiles would be included using the *includeFile* element. The architecture-dependent parts would be included with the *includeFileChoice* element.

The *includeFileChoice* element has a number of *choice* elements, each defining a file to include for a specific architecture. Each *choice* contains an *architecture* element and an *includeFile* element. The *architecture* element holds an object of *archType* type, which is a subset of the string type. The definition of *archType* is given in the previous section for the program schema. The *includeFile* element contains the path of the file to be included if the corresponding *architecture* element matches the chosen architecture.

The third type of content element a *section* can have is the *preproc-objects* element. This element is used to define makefile preprocessor macros and definitions and can even be extended for the definition of other flags or options. The *preproc-objects* element has three parameter elements[1] to help with the formatting of the preprocessor macros and definitions (preprocessor objects) in the makefile. The *startText* element contains a string that leads off the section. The *endText* element deals with the string that ends the section. The *separator* element holds a string that is inserted between the preprocessor objects in a section. The following illustrates the purpose of these elements. The *startText* element contains "`CPPFLAGS = `", and *endText* is an empty string. The *separator* element has a string of " " to produce the space-separated list of macros/definitions.

```
CPPFLAGS = -Dresetjulian -Dtiming=10
```

The *preproc-objects* element also contains a number of *preproc-obj* elements. Each *preproc-obj* has the elements of *name*, *info*, *header*, *value*, *use*, and *requires*. The names of the *preproc-obj* elements in the above example are "resetjulian" and "timing." The *header* element stores the string used for the preprocessor object. This is the actual text displayed for the preprocessor object when its *use* element contains *true*. It is useful to have support for the definition of variable values, so the *value* element was included in the schema. In the above example, "`-Dresetjulian`" and "`-Dtiming`" are each enclosed by *header* tags. The first preprocessor object does not have a *value* element because it is not required to contain a value for definition. The second object

[1]The contents of these formatting elements are included in the generated makefile as they appear in the XML elements.

contains a *value* element of 10. When the *value* element is not needed to display the assignment of a value, it can be omitted from the makefile description file. The *use* element holds either *true* or *false*, and helps to determine whether a preprocessor object is included in the makefile or not.

The *requires* element defines the dependencies of preprocessor objects within a makefile. It consists of a list of *item* elements. Each *item* element holds the name of a preprocessor object that the current preprocessor object is dependent on. For example, if using object A requires that objects B and C are also used, then the *requires* element of preprocessor object A would contain two *item* elements, holding the names of objects B and C. If an object has no dependencies, then there would be no items in its *requires* list.

After all the *section* elements, the last element for the makefile is *conflicts*. This element specifies the conflicts that can occur from using particular combinations of preprocessor objects used within the defined makefile sections. If no such conflicts exist, this element can be omitted from the XML description for the makefile. The *conflicts* element contains a sequence of *conflict* elements, each having a number of *item* elements. The *item* elements hold the names of preprocessor objects. If preprocessor objects A, B, and C should not be used in the makefile at the same time, a *conflict* element must be used to define such a conflict. Within the *conflict* element, there would be three *item* elements used for the names of objects A, B, and C. In such a case, it is fine for one or two of the three preprocessor objects to be used in the makefile, but not all three. Similarly, a conflict defined with two objects implies that only one of the two objects can be used in the makefile at any time. An interesting side-effect of this design appears when a conflict is defined with one preprocessor object. This causes a conflict to occur if the specified preprocessor object is ever used in the makefile, indicating a case that is deemed buggy and should not be used.

3.4 Binary Schema

The binary schema restricts XML description files containing the basic information of the compiled program binaries. It is included in Appendix C as *binary.xsd*. This schema file consists of a top-level *binary* element, along with seven sub-elements that provide the basic parameters for the binary and are used by the user to customize the program execution. These elements are:

name: The name of the binary.

path: The full path of the binary.

info: The description of the binary.

constants: The list of binary constants.

files: The sequence of input/output/in-out files for the binary.

cl-args: Possible command-line switches and arguments.

stdin: Standard input of binary.

Figure 3-3 shows the hierarchy of the binary schema. The * symbol next to the *constant*, *file*, *cl-arg*, and *var* elements indicate that there could be any number of these elements within an XML description file for a binary. The full hierarchy of the *file* and *var* elements are shown in Section 3.5

The *name*, *path*, and *info* elements offer the basic binary information and are treated as string-type elements. The *constants* element is a custom-defined element that consists of any number of *constant* elements. Each *constant* has a preset *name*, *info*, *value*, and *type*. These constants can be used globally by the binary during execution.

The next element is *files*, which is also custom-defined list of elements like *constants*. The *files* element lists all of the I/O (input, output, in-out) files used by the binary. The definition of the *file* element is given by the parameters schema in Appendix D since the *file* is a possible datatype for the parameters and is shared

```
-binary
    -name
    -path
    -info
    -constants
        -constant*
            -name
            -info
            -value
            -type
    -files
        -file*
    -cl-args
        -cl-arg*
            -switch
            -use
            -info
            -var*
    -stdin
        -switch
        -use
        -info
        -var*
```

Figure 3-3: Hierarchy of Binary Schema

by both schemas. The binary schema definition includes the file for the parameter schema, so that various elements defined in the parameter schema can be used by the binary schema. These elements include the *var* element for variables and the eight supported datatypes (e.g. *file*). A *file* has a file-type attribute , along with the parameter sub-elements of *id*, *path*, *info*, *xmlDesc*, and *maxLength*. The possible file types are "input," "output," and "inout" (for those files that are read in from, written to, or both, during binary execution).

The file *id* can be used to synchronize file parameters in various program description files. If a user makes changes to a file variable with a certain file ID, the changes are reflected on all other file variables with the same ID. This is not the same as the input/output logical unit number used in Fortran. The *path* and *info* offer additional facts about the binary. The *xmlDesc* element contains the path for the XML description file of the given file. For example, the program could have an input parameter file that is also encapsulated with an XML description, so that the prototype system developed for this thesis is able to parse this description and allow the user to edit the variable values in the parameter file. Finally, the *maxLength* element constrains the length of the file *path*. This constraint is given in case there is a need to place an upper limit on the length of the file path, as is the case of many operating systems.

The *cl-args* element is a list of *cl-arg* items. Each *cl-arg* defines a command-line option used for the execution of the program, with the appropriate switch and value(s). For each *cl-arg*, the schema includes provisions for a *switch*, *use*, *info*, and any number of *var* elements. The *switch* is the flag or option used in the command line, and the *var* elements are used for the values that follow the switch. The definition of the *var* element is also given by the parameters schema, which is included in the binary schema for the reuse of defined components, as stated earlier. The *use* element can be either *true* or *false* and is used to determine whether the command-line option is used or not.

In the following command, "`-Pprinter-name`" is an example of a command-line option, with the switch being "`-P`" and a variable value being "`printer-name`". If the *use* element for the option were set to *false*, the call to execute the *lpq* binary will not include the option.

```
lpq -Pprinter-name inputfile.txt
```

The last element for the binary is *stdin*, used to indicate the standard input. The *stdin* element can be thought of as just another command-line item, with the switch being "`<`", therefore the schema treats the *stdin* and *cl-arg* elements equally. In the event where the standard input is a parameter file, a file variable is used for the *var* element. Such a parameter file could have its own XML description that allows the user to edit the parameter values using the developed prototype program.

3.5 Parameters Schema

The parameters schema constrains the definition of XML description files for input parameter files used during program compilation and execution. Appendix D includes the parameters schema. The top-most element of this schema is the *paramfile*, which represents an individual parameter file. The *paramfile* has the following elements: *path*, *info*, *startText*, *endText*, and any number of *set* elements.

```
-paramfile
    -path
    -info
    -startText
    -endText
    -set*
        -order
        -info
        -startText
        -endText
        -separator
        -var*
            -type
            -name
            -header
            -use
            -datatype-specific element#
```

Figure 3-4: Hierarchy of Parameters Schema

Figure 3-4 demonstrates the hierarchy of the parameters schema. As with the schemas described previously, elements with a * symbol can occur in any number. The datatype-specific element under *var* element corresponds to the *type* attribute of the parameter variable. For example, if a variable is of type *integer*, then its *var* element must contain an *integer* element. The eight available elements to choose from are: *integer*, *long*, *float*, *double*, *string*, *enumerated*, *uneditable*, and *file*. Datatypes[2] and datatype-specific elements will be explained further in the following subsection.

The *path* element indicates the desired path of the parameter file on the local

[2]The more exotic datatypes, such as *long double*, were deemed not necessary for user-supplied parameters in the proposed schema design.

system. The *info* element holds a string for the description of the parameter file. The *startText* and *endText* elements are used for formatting purposes. They contain strings to be included at the beginning and end of the parameter file. The *set* elements define sets of parameters or variables (referred to collectively as parameters from here on), and are similar in functionality and purpose to the sections of preprocessor objects in the makefile schema. Each *set* is intended to handle a line of parameters as they appear in the parameter file. For example, if a parameter file consists of three lines of parameter values to be read in by the program binary, the description file for the parameter file would have three *set* elements.

Each *set* element contains two informational and three formatting parameter elements, plus at least one *variable* element. The *order* element provides an ordering of the sets to distinguish them from one another. The *info* element is just the standard string-type description for the set. The *startText*, *endText*, and *separator* elements are used in the same way as their counterparts in the makefile preprocessor macros and definitions. Any extraneous text before and after the parameter values on a line (within the same set) are contained in *startText* and *endText*. The parameter files sometimes have extra lines of comments (interspersed between lines of parameters) that are ignored by the program binary. These comments can be contained within *startText* and *endText* as well. The *separator* element holds the separator string inserted between the parameters on each line. Common separators include the space (" "), the comma (","), and the semicolon (";").

The *variable* elements correspond to the parameter variables. Each *variable* has a *type* attribute, which denotes the datatype of the variable. The *type* attribute can only contain one of eight possible strings for the datatypes. The parameter schema supports eight datatypes in total; they are discussed in detail in Section 3.5.1. Each *variable* has four subordinate elements: *name*, *header*, *use*, and a datatype-specific element to describe the variable's value and define any additional constraints for that value.

The *name* and *header* elements are treated as string elements. The *name* element is self-explanatory, while the *header* is used for textual formatting. By default, only the variable values are displayed in the parameter file. However, some encapsulated systems may require variable assignments in the form of “var=value.” This case is supported by the use of the *header*, where the *header* holds the “var=” portion for the variable. The *use* element has a value of either *true* or *false*. This element determines whether a variable value is included in the parameter file or not. The final datatype-specific element is one of eight elements corresponding to the eight datatypes. These elements are: *uneditable*, *string*, *enumerated*, *integer*, *long*, *float*, *double*, and *file*. For more information regarding these datatypes, please refer to Section 3.5.1.

The following two examples demonstrate the use of the *set* and *var* elements. In the first example, the first line of text plus the “`#define `” part of the second line are all contained in the *startText* of one set. The *endText* element is empty, and the *separator* element is not used, since there is only one parameter defined within the set. The *var* element for this parameter is a Boolean-type variable (covered by the enumerated datatype in the schema), with its *header* containing “`MMAX `” and its *value* containing “`TRUE`”. The second set has a *startText* of “`#define `”, an empty *endText* and contains one parameter also. The *var* element for this parameter is a string-type variable, with its *header* containing “`HELLO `” and its *value* containing “`‘‘hello to you’’`”.

```
// some constraint
#define MMAX TRUE
#define HELLO ‘‘hello to you’’
```

The second example below is described by one *set* element. The *separator* of the set is “,” for the comma-separated list of parameters. The *startText* element holds the entire first line, plus the beginning of the second line: “`      parameter(`”. The *endText* element holds the “`)`” from the second line, plus the entire third line. The *set* has two integer-type *var* elements for the two parameters enclosed. The *header* elements for the two *var* elements contain “`mcseg=`” and “`mclen=`”, and their *value* elements are 7 and 2000, respectively.

```
#ifdef coast
      parameter(mcseg=7,mclen=2000)
#endif
```

3.5.1 Parameter Datatypes

There are a total of eight datatypes available to the parameter variables. Four of them are numeric, three are string-related, and the last is a custom datatype for files. The design of these schemas includes specific datatypes in order to allow for the verification of parameter values. Each parameter value can be verified against its own datatype and possible constraints to ensure proper functionality by the executed binary. Such verification would not be possible if all values are treated as string-type values, as in the case of DTD (Section 2.1.2).

Integer, Long, Float, Double

The four numeric types are *integer*, *long*, *float*, and *double*. The *integer* type is for numbers without fractional parts. Negative values are allowed. In most situations, the *integer* type is the most practical. If larger integer values (magnitude-wise) are needed, the *long* type must be used. The *float* type denotes numbers with fractional parts. Since the limited precision of *float* is simply not enough for many situations, a *double* type is provided to allow for double precision values.

The four numeric types each has its own element defined in the parameters schema. Each numeric element has the following parameter elements: *value*, *info*, *units*, and *range*. The *value* element stores the numeric value of the variable. The *info* and *units* elements give the description and unit label of the variable. If a numeric variable is unitless, the *unit* element is left empty. The final parameter for a numeric element is the *range*. The convention used by this system for the *range* is to have a semicolon-delimited (;) list of the legal value ranges. Each range has the form of *lowerValue˜greaterValue*, where the two values are separated by the tilde symbol (˜). The reason for the use of the tilde symbol instead of the dash (–) is to prevent confusion caused by the negative sign. Machine parsing and processing of the range string is much easier when the symbol used for the range (tilde) is distinct from the symbol for negativity (dash). The schema supports the use of any numeric value. Negative and positive infinity are indicated by the strings "NEGATIVE_INFINITY" and “POSITIVE_INFINITY,” respectively. The system does not support the parsing of single values currently, so the definition of a single numeric value within the *range* element would require the value to be both the upper and low limits of the range. If the legal range of a float-type variable is defined to be any negative value less than -10.0, any non-negative value less than 20.0, and the value of 200.0, such a range is captured by the following element.

```
<range>NEGATIVE_INFINITY~-10.0;0.0~20.0;200.0~200.0</range>
```

If there are no constraints on the legal values a numeric variable can have, then the *range* element is omitted from the variable element. The hierarchy of the *integer*, *long*, *float*, and *double* elements are shown in Figure 3-5.

```
-integer/long/float/double
    -value
    -info
    -units
    -range
```

Figure 3-5: Hierarchy of Numeric Datatypes

String

There are three string-related datatypes presented in the parameters schema. These are the *string* type, *enumerated* type, and *uneditable* type. The *string* type is just normal text and can contain any character. The *string* element contains the standard *value* and *info* elements used by all datatype elements, as well as two constraint elements (*minLength* and *maxLength*). The *minLength* and *maxLength* elements contain the integral constraints for the length of the *string*-type variable. This is to offer additional control over the *string*-type variable. If there is no need for length constraints on the variable, one or both of the constraint elements can be left out of the variable description. Figure 3-6 illustrates the structure of the *string* element.

```
-string
    -value
    -info
    -minLength
    -maxLength
```

Figure 3-6: Hierarchy of String Datatype

Enumerated

The *enumerated* type can be considered as a subset of the *string* type, where the allowable values are limited to only a set of strings. Some common examples that require the use of enumerated variables include the suits in a standard deck of cards (diamond, heart, club, spade) and the days of the week (Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday). The *enumerated* element has a *value*, *info*, and a list of *acceptableValues*. The *acceptableValues* element holds a semicolon-separated list of legal values for the enumerated variable. The card suits example is represented by the element:

```
<acceptableValues>diamond;heart;club;spade</acceptableValues>
```

The *enumerated* datatype is designed in this system to cover for the *boolean* datatype. The main reason for this is because there exists no standardized representation of Boolean values across various systems and programming languages. Some encapsulated programs use *true* and *false*; others use *1* and *0*. Some systems are case-sensitive; others are not. The *enumerated* datatype is capable of defining any possible Boolean value-pairs that systems use – be it *true/false*, *T/F*, *hi/lo*, or *on/off*. Thus, *boolean* is not one of the supported datatypes in the schema design.

The hierarchy of the *enumerated* element is shown in Figure 3-7.

```
-enumerated
    -value
    -info
    -acceptableValues
```

Figure 3-7: Hierarchy of Enumerated Datatype

Uneditable

The final string-related datatype is the *uneditable* type. This type is not a standard datatype in programming languages, but was implemented in the schema design. The *uneditable* type is essentially a *string*, except its value is not intended to be altered at any point in time. The *uneditable* element contains only the *value* and *info* elements, and the hierarchy is similar to those of *string* and *enumerated*, without the extra constraint elements.

```
-uneditable
    -value
    -info
```

Figure 3-8: Hierarchy of Uneditable Datatype

File

The *file* element contains a *type* attribute and five other parameter elements. The *type* attribute indicates the file type (input, output, and in-out). This is to give the relation of the file with respect to the program binary, with "inout" representing files that the binary reads from *and* writes to during execution.

The parameter elements for the *file* element are the *id*, *path*, *info*, *xmlDesc*, and *maxLength*. The *path* element is analogous to the *value* element used by the other datatype elements. It is the editable value of the file variable. The *id* element contains an identification for the file. It is useful to be able to link occurrences of a file variable in various locations, and the file *id* provides a mechanism for doing so. For example, a file could be listed in the binary description file, as well as one of the parameter description files (even multiple occurrences within the parameter description file). When the user makes a change to some aspect of the file variable in one occurrence, the other occurrences should also be updated to reflect the change.

It would be very inefficient if the user has to hunt down all these occurrences and make the changes manually. A better solution would be for the prototype program that is processing the description files to make the updates automatically. Therefore, if a file *id* is supplied, this would facilitate the linking process. A simpler alternative exists, where any change made to one file variable is reflected across all file variables with the same path. The disadvantage of this approach is evident in the case where a file has the empty string or null for its path (or some system-specific default). In this scenario, changes made to one variable could inadvertently cause changes to other unequivalent file variables.

If the given file variable has a corresponding XML description file, the path of the description file is given by the *xmlDesc* element. The *xmlDesc* element is left empty otherwise. The final parameter element is *maxLength.* Its use is identical to the *maxLength* of the *string* element. This constraint provides an upper limit on the length for the path of the file.. If there is no constraint on the maximum length of the path, this element is not required.

The *file* element's hierarchy is given in Figure 3-9.

```
-file
    -type attribute
    -id
    -path
    -info
    -xmlDesc
    -maxLength
```

Figure 3-9: Hierarchy of File Datatype

Chapter 4

Prototype System

With a schema design in place to constrain XML files describing the usage of programs, some software is also needed to parse such XML files and display the file contents in an organized manner. This chapter describes the software written as part of this thesis to parse in XML description files conforming to the schemas presented in Chapter 3 and to generate relevant user controls that allow facilitated usage of the encapsulated program.

4.1 Overview

The prototype system is written in Java, and can be run as either a standalone application or an applet. The compiled Java files, along with the necessary JDOM and Apache XML parser libraries, are archived into a JAR file named *prototype.jar*. To run the system as an application, the following command is entered (assuming that the user has Java Runtime Environment installed).

```
java -jar prototype.jar
```

To run the system as an applet, the code to load the applet must be embedded within a webpage. The code below can be placed in the HTML of the webpage to run the applet when the webpage loads.

```
<APPLET CODE="app/Main.class" ARCHIVE="prototype.jar">
```

An interesting side effect resulting from the use of applets is its inability to be shut down like a normal application. A normal Java program exits with the *System.exit* command, which terminates the program. However, when this command is used on Java applets, it causes the browser that loaded the applet to shut down as well. This is because applets are embedded within webpages much like the way images and tables are embedded. An image or table cannot be closed on its own. As part of a webpage, the entire webpage must be closed altogether. Therefore, it is not possible for an applet to close without inadvertently forcing the browser to close as well. A workaround of this is to hide the applet when the user attempts to close it. The system provides a button in the webpage; the visibility of the system GUI window is toggled on and off by pressing on the button. While this does not shut down the applet, it achieves the desired effect.

The system uses Apache's Xerces-J XML parser described back in Chapter 2 to parse all the XML description files. The resulting document object can be accessed and manipulated by the system. JDOM generates a tree-like data structure, with the nodes representing the embedded XML elements within the parsed document. The data from within the element nodes is retrieved and altered by the system via the JDOM API.

Once the JDOM data structure is created, the system can build a Java Swing-based GUI (graphical user interface) using the available data. The various types of description files (program, makefile, binary, parameter file) can be opened independently, or as part of a program. This means that the user can choose to open a makefile XML description file on its own, or jointly with the description file of the program that the makefile is associated with. The system is able to open and support up to one instance of each type of description file during its operation. It should be noted that the system is capable of dealing with files from both the local system and across the Internet. While the paths of the binaries, makefiles, and parameter files must be on the local machine, the XML description files do not have this constraint. The path for an *xmlDesc* element can either point to a local file or an URL for a Web-based source.

4.2 GUI Design

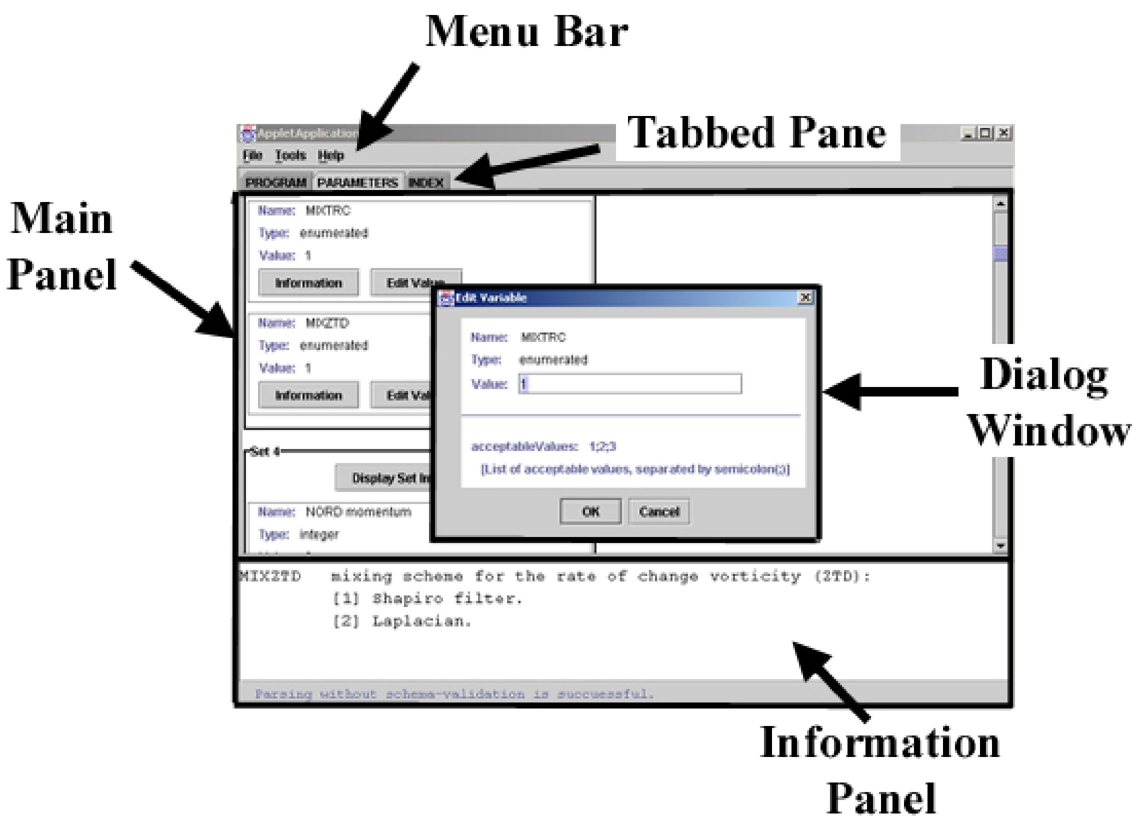
A *graphical user interface* (GUI) offers a pictorial interface to a program. GUIs allow users to operate a program more productively, without having to deal with the low-level details commonly found in text-based user interfaces. GUIs are built from GUI components, which are objects that the user interacts with, via the mouse or keyboard.

The classes used to generate the GUI components for the prototype system are part of the Swing [11] GUI components from the package *javax.swing*. The Swing components are the newest GUI components for the Java 2 platform, and are written, manipulated, and displayed entirely in Java. This is in stark contrast to the original GUI components offered by Java in the *java.awt* package. The Abstract Windowing Toolkit (AWT) components are tied directly to the local platform's GUI capabilities. As a result, AWT-based Java GUIs run on different platforms have different appearances and interaction options (oftentimes referred to as the *look and feel*).

Swing components are considered as lightweight components since they are written completely in Java and are not affected by the GUI capabilities of the local platform. Because Swing components are pure Java components, they offer a greater degree of flexibility and portability than the AWT components.

Figure 4-1 shows the Swing-based GUI of the prototype system. Using various components and features from Swing, the system is able to provide a standardized look and feel across all platforms. The system GUI consists of a menu bar with a main panel and an information panel. The menu bar has three menus: *File*, *Tools*, and *Help*. The *File* menu has the basic system and file operations, such as "Open" and "Close." The *Tools* menu allows the user to parse the description files and modify the schema validation option. The *Help* menu gives a dialog window with a short description of the system.

The contents of the main panel are organized by tabs, based on the type of the parsed XML description file. After the system processes the parsed description files, GUI elements are generated (i.e. labels, buttons, text areas, checkboxes) within



Name: MIXTRC
Type: enumerated
Value: 1

Name: MIXZTD
Type: enumerated
Value: 1

Name: NORD momentum
Type: integer

Name: MIXTRC
Type: enumerated
Value: 1

acceptableValues: 1;2;3
[List of acceptable values, separated by semicolon(;)]

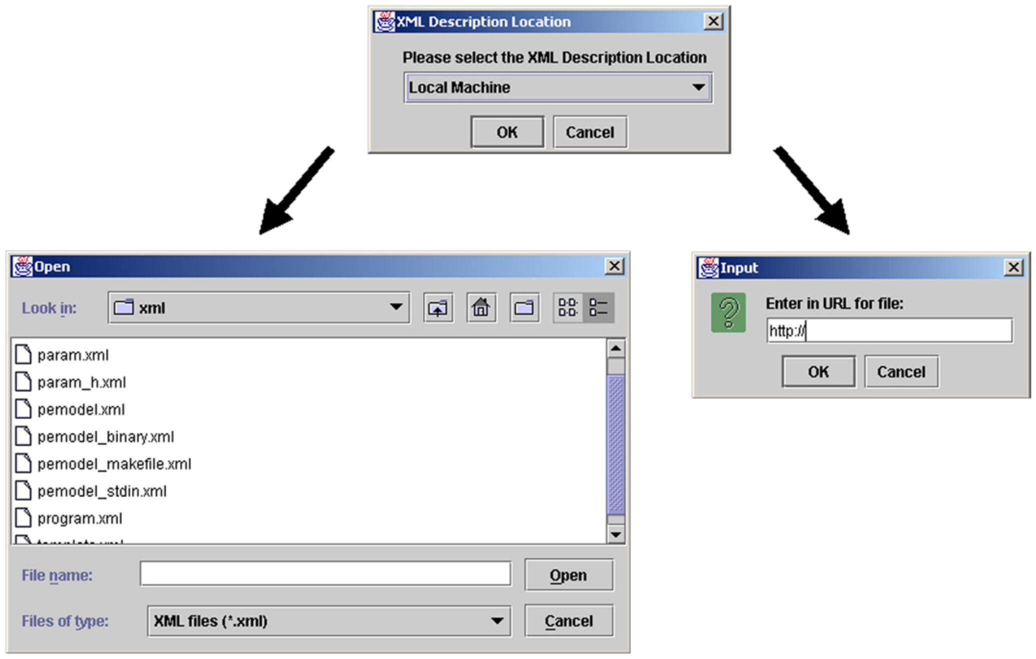
```
MIXZTD   mixing scheme for the rate of change vorticity (ZTD):
         [1] Shapiro filter.
         [2] Laplacian.

Parsing without schema-validation is successful.
```

Figure 4-1: Swing-Based System GUI

the main panel tabs accordingly, to display relevant information and allow for user interaction. The bottom information panel shows the descriptions of various items as the user clicks on the "Information" buttons for these items to learn more about them. Dialog windows are created as needed for the user to edit parameter values or create new ones. Dialogs also provide information to the user and prompt for user selection.

4.3 Functionality

This section contains a walkthrough of the prototype system's GUI components in order to explain the functionality. As mentioned previously, the prototype system is capable of opening XML description files from the user's local system, as well as from remote sources on the Web. The user begins by going to the *File* menu on the menu bar and selecting “Open.” This brings up a dialog prompting the user to choose the source of the file to be opened. The two available options are “Local Machine” and “Web-accessible.” The user selection will then bring up either a *JFileChooser* dialog window for the user to browse the local file system, or a dialog window for the input of a web-based source file’s URL. Figure 4-2 shows these various dialogs.



Please select the XML Description Location: Local Machine

Look in: xml

param.xml
param_h.xml
pemodel.xml
pemodel_binary.xml
pemodel_makefile.xml
pemodel_stdin.xml
program.xml

File name:

Files of type: XML files (*.xml)

Enter in URL for file: http://

Figure 4-2: Dialogs for Opening File

After opening the specified XML file, the system will display the contents of the file in the main panel. For user convenience, the path of the most-recently opened file during the current session is saved. For example, if the user last opened a web-based file with the URL of *http://www.files.com/program.xml*, the next time the user decides to open another web-based file, that path will already be present in the dialog. This feature was added since users commonly open files from the same directories (having all the input files stored in the same directory). In addition, reopening the same file causes all the values to reset. Sometimes it is just too tedious for the user to change the parameter values one by one to restore all values back to the default; having this reset capability will facilitate the process.

After opening the file, the prototype system must then parse the XML description in order to generate all the corresponding GUI elements that allow the user to customize the program's compilation and execution parameters. This is accomplished by selecting the "Parse XML" item from the *Tools* menu. Before parsing, the user has the option to enable or disable the schema validation option of the system. By default, the system parses all XML files without schema validation. This is assuming that the user knows better than to have the system parse illegitimate XML files that are unsupported. The option menu is shown in Figure 4-3. It is safer to have the schema validation capability turned on, in case the user is unsure of whether the source XML file conforms to the XML schema design proposed in the previous chapter. Parsing is slightly faster without schema validation.

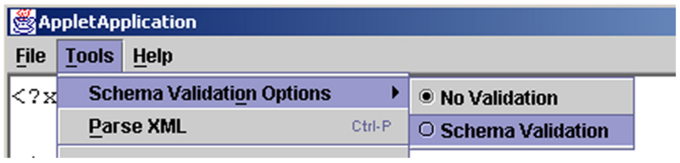
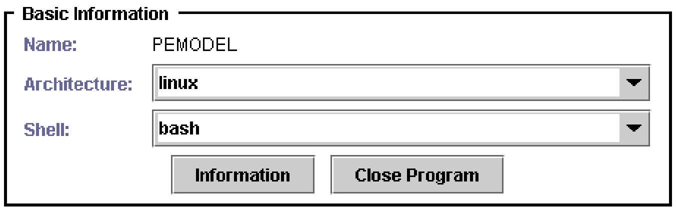


Figure 4-3: Options for Parser Validation Against Schema

The resulting GUI generated by the system depends on the type of description file parsed. There are four total description file categories (program, makefile, binary, parameter file); the user interface for each of these categories is discussed in detail in the following subsections.

4.3.1 GUI for Programs

When the XML description file for a program is parsed, the GUI generates a *PROGRAM* tab, with the relevant information about the program displayed, which serves as a gateway for accessing descriptions of makefiles, binaries, and parameter files associated with the current program. The *PROGRAM* tab has four main sections, which are mainly consistent with the major parameter elements of the *program* element from the XML description file. The "Basic Information" section, given in Figure 4-4, provides the information regarding the program name, description, architecture, and desired shell program. From the list of available architectures and shells given in the program description file, the GUI creates two pull-down lists for the user to choose from. The "Information" button displays the program description in the bottom informational panel of the window. The "Close Tab" button deletes the tab and closes the current program description file.


Basic Information

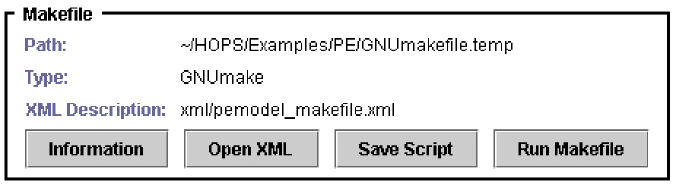
Name: PEMODEL

Architecture: linux

Shell: bash

Figure 4-4: Basic Information Section in *PROGRAM* Tab

The "Makefile" section, as seen in Figure 4-5, shows the path of the makefile for the current program on the local system and the type of make tool to be used on the makefile. If there is an XML description file available for the makefile, the path for that description file is given and the "Open XML" button is provided. When the makefile does not have a description file, only the other three buttons will be visible.


Makefile

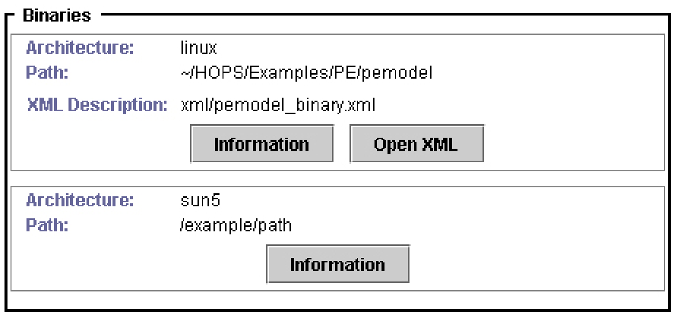
Path: ~/HOPS/Examples/PE/GNUmakefile.temp

Type: GNUmake

XML Description: xml/pemodel_makefile.xml

Figure 4-5: Makefile Section in *PROGRAM* Tab

The "Open XML" button allows the user to open the XML description of the makefile in conjunction with the program description. This creates an additional *MAKEFILE* tab, which displays information about the makefile and allows the user to generate a customized makefile. The parsing of the makefile description will incorporate the schema validation setting at the time, and the tab for the makefile will be generated without displaying the associated XML contents first. This is because the makefile description is opened as part of the current program. The "Save Script" button allows the user to choose the path of the generated shell script for compiling


Binaries

Architecture: linux
Path: ~/HOPS/Examples/PE/pemodel
XML Description: xml/pemodel_binary.xml

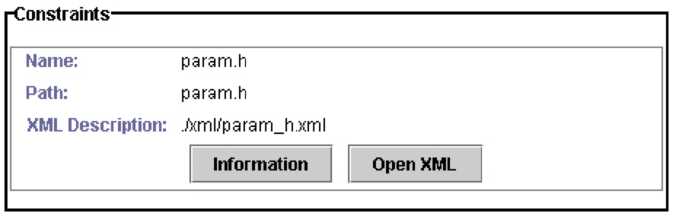
Architecture: sun5
Path: /example/path

Figure 4-6: Binaries Section in *PROGRAM* Tab

the program source code using the given make tool and makefile. Finally, the "Run Makefile" button allows the user to execute the saved script.

The "Binaries" section gives a listing of all binaries available for the opened program. The GUI displays the path of each compiled program binary, as well the binary's XML description file if it is available. A screen shot of the "Binaries" section is provided in Figure 4-6. As with the "Makefile" section, the user can open a binary's description file automatically by clicking on the "Open XML" button. This causes the system to associate the binary description with the current program description. Thus the binary is not treated independently and is considered as being part of the program. Opening and parsing the binary description will add five tabs to the GUI. Section 4.3.3 elaborates on these tabs. In the provided example, the binary corresponding to the "sun5" architecture does not have a description file.

The last section in the *PROGRAM* tab is "Constraints." The system creates a bordered box for each constraint item listed in the program description file. These boxes are all set within the main "Constraints" section. Used in the context of the program, constraints are simply additional files required during the make process that contain build parameters. Each constraint box gives the name and path of the constraint file, as well as the path of the constraint file's XML description, if available, along with a corresponding button to open the description.

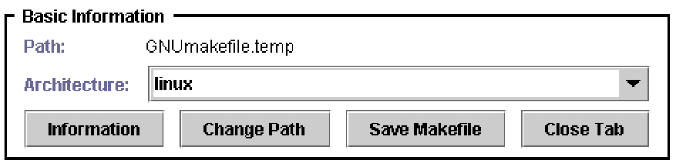


Name: param.h
Path: param.h
XML Description: ./xml/param_h.xml

Figure 4-7: Constraints Section in *PROGRAM* Tab

4.3.2 GUI for Makefiles

Opening and parsing a makefile description file will prompt the system to create a tab named *MAKEFILE*. This tab contains the basic information and elements for the makefile. The sections of the makefile (corresponding to the *section* elements from the makefile schema) are each given a bordered panel in the tab. Each section corresponds to a file to be appended to the makefile, a file from a list of choices to be appended to the makefile, or a list of preprocessor macros and definitions to include in the makefile. The details of the various types of makefile sections were discussed back in Section 3.3. After the sections, the *MAKEFILE* tab also includes a list of the defined conflicts for the preprocessor objects in the makefile.



Path: GNUmakefile.temp
Architecture: linux

Figure 4-8: Basic Information Section in *MAKEFILE* Tab

Figure 4-8 shows the portion of the *MAKEFILE* tab for basic information. The path and architecture of the makefile are given. The user can change the path of the makefile on the local machine using the “Change Path” button. The system saves the generated makefile to the specified path when the “Save Makefile” button is pressed. “Close Tab” simply closes the current instance of the program makefile.

The user can change parameters within the current makefile through the GUI components provided for each section of the makefile. Figure 4-9 displays the contents

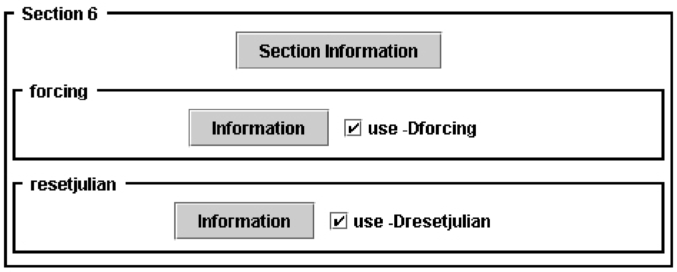
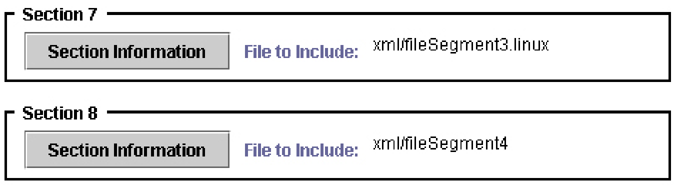


Figure 4-9: Section for *preproc-objects* Element in *MAKEFILE* Tab

of a makefile section for preprocessor objects. The preprocessor objects used in the section are given in smaller bordered boxes. Each preprocessor object box contains an "Information" button, as well as a checkbox used for the inclusion of the preprocessor object in the generation of the makefile. The checkbox maps directly to the *use* element within each *preproc-obj* element, as presented in the makefile schema. By toggling a preprocessor object's checkbox, the user is able to change the value of the *use* element (can be either true or false). All the other aspects of the *preproc-obj* element, such as *value* and *requires*, are not included in the GUI, because these aspects are fixed and are extraneous for the user. Their inclusion in the GUI would only add more confusion. The preprocessor object dependencies are checked by the system before saving the makefile. This is to ensure that all necessary preprocessor objects have their checkboxes toggled.

There is really no difference in the GUI between the makefile sections for the *includeFile* elements and the *includeFileChoice* elements. To the system, a file is included in both cases. In the example shown in Figure 4-10, the first section corresponds to an *includeFileChoice* element, while the second corresponds to an *includeFile* element. The included file in the first section can change depending on the selected architecture for the makefile, while the same file is included in the second section for any of the architectures. The file *fileSegment3.linux* is included for the first section in the figure, corresponding to the selected "linux" architecture. If another architecture is selected, such as "sun5," then another file is included in place of *fileSegment3.linux*.


Section 7

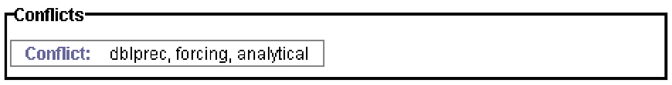
Section Information | File to Include: xml/fileSegment3.linux

Section 8

Section Information | File to Include: xml/fileSegment4

Figure 4-10: Sections for *includeFile* and *includeFileChoice* in *MAKEFILE* Tab

The "Conflicts" section of the *MAKEFILE* tab lists all the conflicts defined in the makefile description file. Each *conflict* element from the description file has its own bounded box in this section. The names of the preprocessor macros and definitions that cause the conflict are listed in the box. Therefore, if a box exists in this section with the text "`Conflict: A, B, C`," then a conflict exists for the encapsulated system when preprocessor objects A, B, and C are used at the same time. The prototype system checks to see that no conflicts exist for the objects in the current makefile setting before saving the makefile. The example given in Figure 4-11 shows that all three of the *dblprec*, *forcing*, and *analytical* preprocessor objects cannot be used in the makefile at the same time.


Conflicts

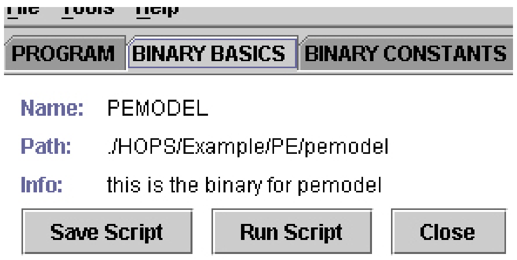
Conflict: dblprec, forcing, analytical

Figure 4-11: Conflicts Section in *MAKEFILE* Tab

4.3.3 GUI for Binaries

The data in the XML description file for a compiled binary is presented in five separate tabs: *BINARY BASICS*, *BINARY CONSTANTS*, *BINARY I/O FILES*, *BINARY COMMAND-LINE ARGUMENTS*, and *BINARY STDIN*.

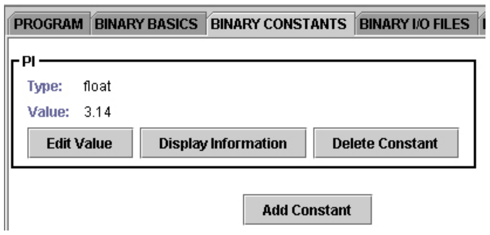
The *BINARY BASICS* tab contains the basic information about the binary and offers features for the user to save and run a shell script to execute the program binary. Textual GUI elements in the tab provide the name, path, and description (info) of the binary. As seen in Figure 4-12, there are two buttons to generate and



Name: PEMODEL
Path: ./HOPS/Example/PE/pemodel
Info: this is the binary for pemodel

Figure 4-12: Contents of *BINARY BASICS* Tab

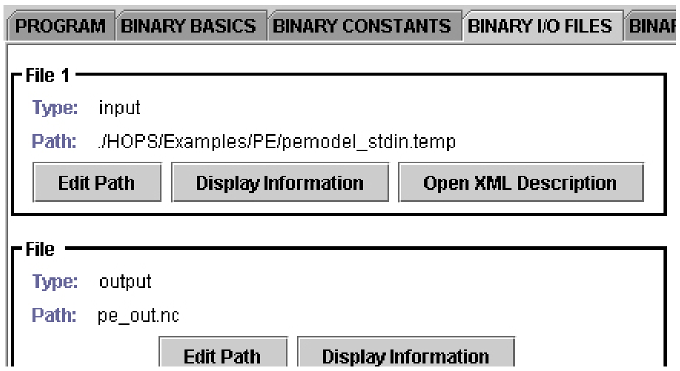
run the binary execution script. A third button closes the binary description file and removes the five binary-related tabs.



PI
Type: float
Value: 3.14

Figure 4-13: Contents of *BINARY CONSTANTS* Tab

The second tab generated from the binary description file is *BINARY CONSTANTS*. This tab provides a list of all the constants defined for the binary. Each constant has its own bordered box, with the relevant information and options for user interaction. The name of the constant is given in the title of the border. Figure 4-13 shows the *BINARY CONSTANTS* tab with the PI constant defined. Each constant has a type and value, both shown as *JTextArea* objects. The user can click "Edit Value" to redefine the value of the constant. Doing so brings up another dialog window that allows the user to change the current value of the constant. Before the system commits the change, the datatype of the user-modified value is checked against the constant's predefined datatype. The user also has the option to define additional constants or delete them as needed. All this functionality is useful only if the binary supports the redefinition of built-in constants and the addition of user-defined ones. The current implementation of the system does not use these constants in any way during binary execution. The constants are shown to offer an idea of a possible GUI design.

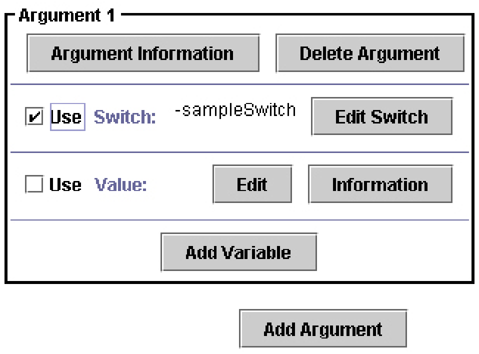


File 1
Type: input
Path: ./HOPS/Examples/PE/pemodel_stdin.temp

File
Type: output
Path: pe_out.nc

Figure 4-14: Contents of *BINARY I/O FILES* Tab

The *Binary I/O FILES* tab lists all the input and output files used by the program binary during execution. Figure 4-14 includes two file listings. Each file entry corresponds to the definition of the *file* datatype element in the parameters schema (Section 3.5). The GUI displays the type and path of each file listing and also provides buttons for the user. For each file, the user can choose to edit the path, or open the XML description of the file, if the *xmlDesc* element of the file has been defined in the description. The file *id* element that is used for linking is shown as part of the border title. In Figure 4-14, an input file with an available XML description and an *id* of "1" is shown at the top. The second file listing is for an output file that has an undefined file *id* and no available XML description file.



Argument 1
[✓ Use] Switch: -sampleSwitch
Use Value:

Figure 4-15: Contents of *BINARY COMMAND-LINE ARGUMENTS* Tab

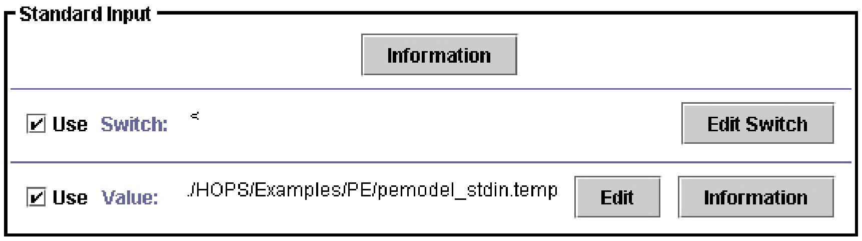
Figure 4-15 shows the contents of the *BINARY COMMAND-LINE ARGUMENTS* tab, having only one command-line argument. Each listed command-line argument (*cl-arg* element in the binary description file) and its associated switch and variables values are enclosed within a box. The values for the switch and variables can all be modified. A *Use* checkbox precedes each switch and variable entry. The system uses the checkbox to determine whether the command-line argument (its switch and values) is included in the call for the execution of the binary. The checkbox corresponds to the *use* element within each *cl-arg* element of the binary description file. A checked box maps to a value of *true*; an unchecked box maps to *false*.

There is an important distinction between the *Use* checkbox for the switch and the *Use* checkbox for the variables. Each of the available variable values for a command-line argument can be included by checking (toggling on) its checkbox. The *Use* checkbox for the switch acts as the master checkbox for the entire command-line argument. When the checkbox of the switch is unchecked, it is assumed that the command-line argument will be omitted, and the switch and all corresponding variable values are not used (regardless of the state of the variable checkboxes). This is better explained through the following command:

```
foo -test 12 39
```

If the above command is for executing a program binary, then "`foo`" would be the binary name and "`-test 12 39`" would be a command-line argument ("`-test`" is the switch, "`12`" and "`39`" are its two variable values). The above command is generated by the prototype system only if all three checkboxes for the switch and two variables are toggled on. If the user unchecks only the checkbox for the "`-test`" switch, then the system would omit the switch and both of the variable values as well, even though their checkboxes have not been unchecked yet.

The user can choose to execute the binary with only the "`-test`" switch and no trailing values (`foo -test`) by selecting only the checkbox of the switch and leaving the checkboxes of the variables unselected. In the case where a program binary is executed with a string of argument values and no switch (e.g. `foo 12 39`), the value of the switch used by the command-line argument would be the empty string. The



Switch: <

Value: ./HOPS/Examples/PE/pemodel_stdin.temp

Figure 4-16: Contents of *BINARY STDIN* Tab

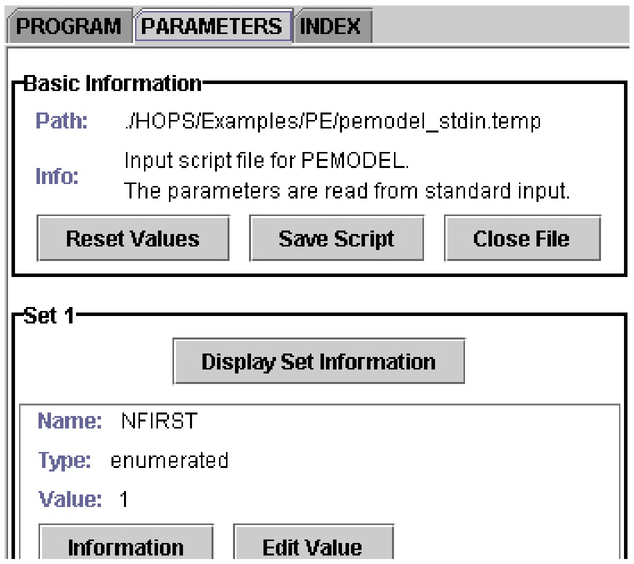
only difference between this example and the "`foo -test 12 39`" example is the value of the switch.

The user has the option to include additional variables for each command-line argument through the "Add Variable" button. Each added variable has its *Use* checkbox toggled on by default. There is no option to delete a variable since the desired result can be achieved by unchecking the checkbox for the variable. The system also supports the functionalities of adding or deleting command-line arguments through the "Add Argument" and "Delete Argument" buttons. This is the only method for capturing the standard output and standard error of the binary, since the current schema design only supports the standard input stream. These buttons will be phased out once the binary schema incorporates standard out and standard error.

The *BINARY STDIN* tab provides the details of the standard input used by the program binary. From the system's point of view, the standard input portion of a command to execute a binary is just another command-line argument with "<" as its switch and having only one variable. Therefore, the user interface for the standard input is similar to the interface for a command-line argument. This is evident from comparing Figure 4-16 with Figure 4-15 shown previously. Furthermore, the *BINARY STDIN* tab does not allow the user to add additional arguments or variables or delete them.

4.3.4 GUI for Parameter Files

When the prototype system opens and parses a description file for a parameter file, two tabs are created initially – *PARAMETERS* and *INDEX*. The *PARAMETERS* tab displays all the major components of the top-level *paramfile* element from the description file (i.e. path, sets of variables). Figure 4-17 shows an example of the *PARAMETERS* tab.



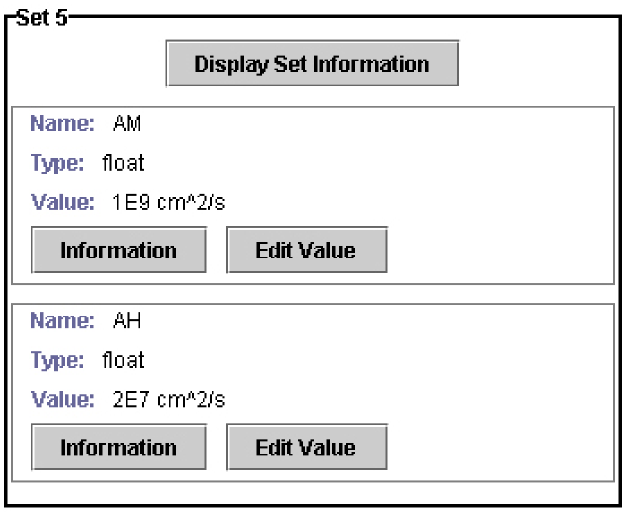
Path: ./HOPS/Examples/PE/pemodel_stdin.temp
Info: Input script file for PEMODEL.
The parameters are read from standard input.

Name: NFIRST
Type: enumerated
Value: 1

Figure 4-17: Contents of *PARAMETERS* Tab

At the top of the tab is a section displaying the parameter file's basic information and also providing buttons for basic operations. The full path of the parameter file and its description are given as textual GUI elements. The values of the parameter file can be reset to the default ones provided in the original description file by clicking on the "Reset Values" button. This is a much more convenient approach than having the user make all the changes manually. The "Save Script" button prompts the system to save a copy of the parameter file with the current parameter/variable values to the specified local path. "Close File" simply closes the current instance of the parameter file and removes all tabs in the GUI relating to the parameter file.

Following the *Basic Information* section, all subsequent sections in the tab correspond to the defined *set* elements from the description file. Each *set* contains a


Set 5

Display Set Information

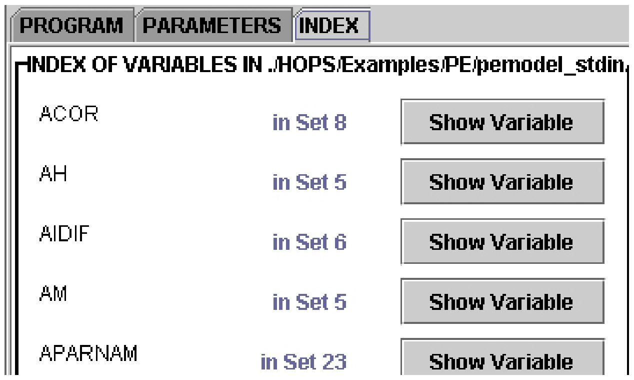
Name: AM
Type: float
Value: 1E9 cm^2/s

Information Edit Value

Name: AH
Type: float
Value: 2E7 cm^2/s

Information Edit Value

Figure 4-18: UI Representation for *set* Element


PROGRAM PARAMETERS INDEX

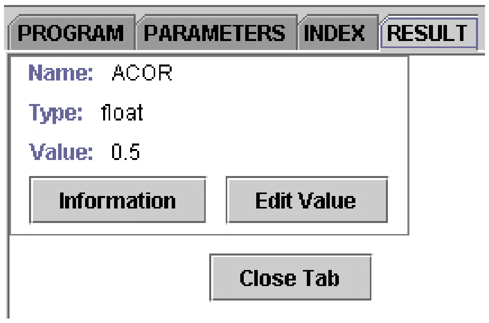
INDEX OF VARIABLES IN ./HOPS/Examples/PE/pemodel_stdin

ACOR	in Set 8	Show Variable
AH	in Set 5	Show Variable
AIDIF	in Set 6	Show Variable
AM	in Set 5	Show Variable
APARNAM	in Set 23	Show Variable

Figure 4-19: Contents of *INDEX* Tab

number of variables, along with their values and constraints. The section for a sample *set* is shown in Figure 4-18. Each variable within the *set* is listed with its name, datatype, and value within a smaller bordered box. The user can modify the value of each variable with the "Edit Value" button. Clicking this button brings up a dialog window that allows the user to make the appropriate changes and displays any constraints defined for the variable. The modified value is checked against the datatype of the affected variable and any possible constraints it has. The system does not update the user modification unless none of the constraints are violated.

As seen in Figure 4-19, the *INDEX* tab provides an index listing of all the variables



Name: ACOR
Type: float
Value: 0.5

Figure 4-20: Contents of *RESULT* Tab

within the current parameter file. The variable names are listed alphabetically, along with the *set* that the variable belongs to. This tab offers an alternative approach for editing variable values in the parameter file. It could be time consuming to search through tens or hundreds of sets for a specific variable in the *PARAMETERS* tab. The index allows the user to browse through the variables based on their names. When the user clicks on the "Show Variable" button, the system generates a third tab named *RESULT*. This extra tab allows the user view the details of the variable and make changes.

The *RESULT* tab brings up the details of the selected variable. The variable is displayed as it appears in the *PARAMETERS* tab, and any change made to the variable in either location affects both locations. The *INDEX* and *RESULT* tabs are meant to let the user search for and modify the value of a variable quickly.

Chapter 5

Initial Results

The XML schema design and system implementation were tested initially with the Primitive Equation (PE) Model binary of the HOPS system [30], which was described in the introduction. PE Model is at the heart of the HOPS forecasting system, and is used to predict the ocean state using state variables such as temperature, salinity and velocity. PE Model reads in NetCDF [14] files and runtime parameters from standard input, and has no command-line arguments. During execution, PE Model outputs to multiple NetCDF files and the standard out and standard error streams.

Many of the decisions made for the schema design were influenced by the PE Model program. As the schema underwent several iterations of improvement, design revisions were based on PE Model, while keeping the elements within the design as general as possible (e.g. command-line arguments) – paving the way for adapting the schema design and prototype system to other applications in the future. As a result, the initial testing was performed with the description files of the encapsulated PE Model. The system enabled the user customization of build-time and runtime parameters and successfully compiled and executed the resulting PE Model binary.

5.1 Writing XML Description Files

XML description files were written for the PE Model program, makefile, binary, runtime parameters provided through standard input (e.g. *pemodel.in*), and build-time parameters defined in a file that always has the name *param.h*.

A top-level program description file was written, with the definitions of the supported architectures for PE Model, and possible shell programs used for scripts. The program description also points to the associated description files of PE Model's makefile, build-time parameter file *param.h*, and compiled binary.

The makefile description file defines the various preprocessor macros and definitions and sections embodied in the makefile. The preprocessor objects are included directly in the description file itself, whereas the sections are included as file fragments to be appended to the makefile. There were four file fragments to be included – two were platform-independent, two were not. For the platform-dependent portions of the makefile, eight versions (corresponding to the eight platforms supported by HOPS) were saved in separate files. The compilation process also requires an input parameter file, *param.h*, which was encapsulated as the *param_h.xml* description file.

The usage of the compiled binary was defined in a binary description file. The runtime parameters of the PE Model binary are provided through the standard input as a parameter file (e.g. *pemodel.in*). This file was also encapsulated in an XML description file, *pemodel_stdin.xml*.

To facilitate the writing of description files, a Java tool was written to check for the well-formedness and schema conformity of the description files. The tool takes in a file, accessible locally or on the Web, and a Boolean argument for schema-validation. An argument of *false* will disable the schema-validation feature, and the tool will only check for the well-formedness of the input file. Otherwise, the tool validates the input file against the declared schema document(s) within the input. The following command demonstrates the usage of the validation tool.

```
java checkXML path/URL validation-boolean
```

By having the description files for all usage aspects of PE Model – from compilation to execution – the encapsulation of the program was completed successfully.

5.2 Using the XML Description Files with the Prototype System

The testing was performed on a Dell laptop computer running Linux. The architecture selected from the program description during testing was "linux" to match the system architecture. From there, the makefile description was processed and various file fragments and preprocessor objects were pieced together to generate the desired makefile automatically. The file fragments corresponding to the Linux platform were included by the system for the makefile automatically, due to the architecture selection of "linux."

The graphical interface components that were generated from parsing these various description files are shown in the various figures of the previous chapter. The source of all these figures was the prototype system processing the encapsulated PE Model binary.

Through the system GUI, the appropriate makefile preprocessor macros and definitions were selected for inclusion in the final makefile. Because the compilation process for PE Model also requires build-time parameters defined in *param.h*, the system first had to process the XML description of this parameter file and generate the file (*param.h*). The prototype system was able to use the generated makefile and build-time parameter file to compile the PE Model source files and build the program binary for the Linux platform automatically.

The PE Model binary has a series of runtime parameters read in from standard input that the user needs to specify before executing the program binary. The input parameter data is provided to the binary in the file *pemodel.in.* Figure 5-1 shows an excerpt of this file. A parameter XML description file, based on the values and types of these parameters, was written for *pemodel.in.*

After validating the XML description files against their respective schemas, the prototype system was able to process the information about the binary and its parameters for display by the GUI. The system presented the contents of the runtime parameter file, which oftentimes can be cryptic and confusing for users to modify, in

```
1    NFIRST  NLAST  DOSTART  NNERGY  NTSOUT  NTSI  NMIX   NCON  NTDGN
1 672 8919.00 92 48 1 10 0 0
2    DTTS    DTUV     DTSF  (seconds)
900 900 900
3    MIXVEL MIXTRC MIXZTD (mixing scheme: momentum/tracers/vorticity)
1 1 1
4    NORD NTIM NFRQ (momentum, tracers, vorticity, and transport)
2 1 1 4 5 1 2 1 1 4 1 0
5    AM      AH
1E9 2E7
 . . . . .
39   TIDEBOX (a80): input ASCII file defining tidal regions.
/dev/null
99   END of input data
```

Figure 5-1: Subset of Parameter File Generated Automatically by System

an organized graphical manner that is easily understood by users. Instead of having to edit the parameter file directly, the parameter values were updated through the GUI. All modifications were checked for legitimacy before the system generated the parameter file (*pemodel.in*) automatically and proceeded to execute the PE Model binary using the parameter file as the standard input.

A subset of the PE Model runtime parameter file generated by the system has already been shown in Figure 5-1. The entire parameter file includes 39 total *cards*, which are analogous to the *sets* defined in the parameters schema. Each *card* contains the group of variables related to a specific aspect of the PE Model binary simulation. For instance, *Card 10* deals with the tidal mixing variables used during the execution of the binary.

Each card corresponds to two lines of output in the parameter file. The first line is ignored by the system and is used for commenting purposes, in order to make the generated file legible for the human user. The second line contains the actual parameter values used for the binary execution. Therefore, the PE Model binary ignores all the odd-numbered lines (with the exception of the very last line). The parameter values are separated by white spacing and appear on the even-numbered lines. The last line beginning with “99” signifies the end of the parameter file for the binary. Even though this is an odd-numbered line, it is not ignored by the binary.

5.3 Additional Issues

5.3.1 Special Characters Used in XML

The XML specification defines certain special characters, such as the ampersand (&) and left angle bracket (¡), that may appear in their literal form only when they are used for delimiters for markup or other special purposes[1] To use these special characters elsewhere in an XML document, they must be escaped using numeric character representations or strings. If these characters are not escaped in the writing of the XML description files, the XML description files cannot be parsed correctly by any parser. The following shows some common special characters and their respective escape strings used in XML documents.

```
& -- &amp;

< -- &lt;

> -- &gt;

' -- &apos;

`` -- &quot;
```

5.3.2 Makefile and Ant

The current approach for the encapsulation of the program makefile treats its contents as a series of included file fragments and preprocessor macros and definitions. All of the shell-based commands within the makefile are placed in the file fragments to be included, and the description file has no knowledge of these shell commands. A possible improvement to this approach would be the use of Apache Ant [1].

[1]Within comments, processing instructions, or CDATA sections.

Ant is being developed by the Apache Software Foundation. It is a build tool based on Java, which gives Ant the ability to be cross-platform. Instead of using the shell-based commands like standard makefiles, Ant uses XML-based configuration files. The XML build file itself describes a project, with a number of targets. Each target consists of a number of task elements and can have dependencies on other targets. For example, a "init" target contains tasks to create several directories, and the "compile" target depends on "init." If the user tries to run "compile," Ant checks to see that "init" is run before "compile" to satisfy the dependency. The target can be thought of as some sort of subroutine, with the tasks being commands within the subroutine.

Ant already has many predefined categories of tasks. Some of these categories include archiving (e.g. creating .zip, .cab, .jar files), execution (e.g. system commands), and file tasks (e.g. copy, delete, remove, etc.). If the user wishes to define a new task, a Java class that extends the Ant Task class must be created.

While Ant seems promising, writing the build files in XML could be somewhat awkward and tedious, especially for larger software projects with many files, definitions, and dependencies. Currently, there are some tools being developed, such as Ant Factory [2], to generate Ant files automatically. Such tools could be helpful when trying to generate thousands of lines of XML used in Ant configuration files.

Chapter 6

Conclusions

6.1 Conclusions

In this thesis, an XML-based approach for the encapsulation of legacy binaries is presented. For a program and its associated components, such as the makefile, binaries, and parameter files, it is possible to create XML description files that conform to the schema design. These schema-validated XML description files will provide a readable standard for the computing of binaries and free the users from the constraints of specific platforms and hardware setups. Through the GUI, the user can control and customize all aspects of compilation and execution for the encapsulated program. As long as a program's source files and XML description files are available, the user is able to compile and execute the program on his local machine.

A prototype system was implemented and had the capability to generate a graphical user interface displaying the relevant program information, along with various compile-time and run-time components. With the appropriate parameter files in place, the system is also able to compile and execute the encapsulated program, using the GUI through the entire process. The HOPS PE Model program was encapsulated effectively using the proposed schema design and prototype system. The GUI allowed for user customization of program settings and build-time/run-time parameters, and also validated the value changes for potential violations of range and datatype constraints before generating parameter files and executing a binary. PE Model was compiled from its source files on a Linux-based laptop computer and the resulting binary was run on the same platform with the run-time parameters as standard input.

6.2 Future Research

With the proposed schema design and prototype system in place, there are many opportunities for improvements and extensions. The main focus of any future work should be to extend the usability of the schema design and to improve the functionality of the system GUI.

6.2.1 Schema Extensions

In order to facilitate the extension of the proposed schema design to XML descriptions of programs from other systems (non-HOPS) later on, the schemas could be extended in several areas. There are other lesser-used variable datatypes that are not supported by the current design. An example of this is the array-type variable. Currently, the system can only represent a sequence of values from arrays or lists as a string of characters (using the *string* variable). While this is a completely legitimate adaptation of the *string* datatype, there is no way to check the validity of the individual array values using the existing implementation.

Another area for extension is the listing of available architectures, makefile types, and shell programs. The current lists are based on the requirements of PE Model, and are not necessarily inclusive of all cases encountered in other systems. For example, the architecture list currently does not include any Windows or Mac operating systems.

The elements associated with the shell program should be expanded to include the path of the shell as well. The current implementation assumes that all the shell programs are available in the standard /usr/bin directory of the system. However, this is not necessarily the case for all systems. Therefore, it would be beneficial to associate each shell program with the path for its executable as well. This can be done by defining each shell element to include a *path* element.

A point brought up during the discussion of the binary schema was the inclusion of the parameter schema in order to reuse defined elements from the parameter schema (e.g. *var* and datatype elements). It might be better to define these elements in a separate schema for common datatypes, so that the binary and parameter schemas are kept independent of each other.

Finally, the binary schema should be expanded to include support for standard output (stdout) and standard error (stderr). The present schema design only covers for the standard input stream during the execution of a program binary. It does not handle the standard output and error streams explicitly. Of course, the three standard streams can all be defined as normal command-line arguments in the binary description file, which is why the option to add command-line arguments in the GUI was implemented, but this approach is less thorough. The standard output and error streams would have predefined switches of ">" and "&>", respectively, for the C family of shells. The standard streams would need to be included on the command-line, in the order of standard input, standard output, and standard error. The Bourne family of shells uses a different syntax.

6.2.2 System/GUI

As with any GUI-based system, the usefulness increases with the level of convenience offered by the GUI. The prototype system already implements many features designed to make the user's life easier and expedite the customization of the compilation and execution processes for an encapsulated program. However, there are many more possibilities for future development.

Even though the current implementation is able to process different types of description files at the same time (i.e. the description files for a program, its makefile, and its binary), each type of XML description file is still handled somewhat separately by the system. An inherent advantage of this approach is that the user can choose to view just the details of a specific component's description file, without having to deal with the description files for its related components. The user can customize the values within a parameter file on its own, without having to open up the description files for the program and binary with which the parameter file is associated.

This approach does make integrating and coordinating the various GUI components harder to do. For instance, the architecture definitions in various locations must be coordinated by the system, so that a change in one location affects all locations. When the user changes the architecture selection in the *PROGRAM* tab, the corresponding selection in the *MAKEFILE* tab must change also. Otherwise, the user would be forced to make such changes manually, and the system becomes more of a nuisance than an useful tool. Incorporating such coordination between GUI components introduces a lot of complexities into the system. Perhaps the system GUI could be organized in a different manner for major revisions in the future – one that is less cumbersome for the user.

Also, the current GUI cannot make universal changes to file-type variables with the same ID yet. Even though the schemas already support this feature of linking files with the same file *id*, the GUI does not yet implement such functionality. When the user changes the path of a *file* variable, all other occurrences of that *file* variable within the XML descriptions of the same program will not change automatically. The user must update each instance of the variable manually.

Lastly, it would be useful to have some conversion utility in place for variable units, such that the user's numeric inputs in unacceptable units are converted to inputs with units allowed by the binary. For example, if an encapsulated program requires a variable to be expressed in meters, values given in centimeters or inches could be converted by the system automatically. The user would save time and would not need to worry about having to do all such conversions manually. This improves the usability of the system and increases its robustness.

6.2.3 Workflows and Grid Computing

The current implementation of the system deals with programs individually. A possible extension of the system would be to introduce the concept of program workflows, similar to that of the problem-solving environments introduced in Section 1.4. The work performed by many scientific systems oftentimes require chaining several different programs together. The output of some programs could be fed in as input for other programs. This can be achieved if a program workflow composition tool is added to the system GUI. The schema design must also be extended to introduce the concept of program workflows, with various programs input and output requirement definitions.

While the system is certainly useful as a Web-accessible applet that is capable of compiling and running a program on the user's local machine, a step must be taken to handle programs located on remote machines as well. Many scientific applications require dedicated workstations or even supercomputer capabilities not available on the ordinary machine. By placing the encapsulated program on remote machines with the appropriate setup and computing power, an user will be able to run such programs and will no longer be constrained by the computing resources available locally.

The eventual goal of this project is to incorporate grid computing technology to enable remote computing. Let us suppose that there is an encapsulated simulation program, requiring a lot of resources that the user has remote access to. The user can use the XML descriptions of the program and the prototype system to compile and run the program. By using grid computing technology, the system can compile and run the program on specific remote machines with the proper resources, and provide the program results back to the user. In this scenario, the user never has to compile and install the program on his/her local machine.

Grid computing technology is associated with the discovery and management of computing resources available across a heterogeneous network of computing infrastructures. There are existing toolkits that provide web portals [5, 6] to Grid infrastructures. The prototype system can be distributed by the web portal as a Java applet – one of the primary reasons why the system was implemented as an applet as well. Included with the portal would be the functionality for the user to discover and manage computing resources available for scientific computations and simulations. Thus, the user can customize the parameters for remote program execution, then utilize the Grid infrastructure to run the binaries remotely.

Bibliography

[1] The Apache Ant project. http://ant.apache.org.

[2] Ant Factory. http://sourceforge.net/projects/antfactory/.

[3] Common Object Request Broker Architecture. http://www.corba.org.

[4] W3C Document Object Model. http://www.w3.org/DOM/.

[5] DOE Science Grid: Grid Portal Development Kit (GPDK). http://doesciencegrid.org/projects/GPDK/.

[6] NPACI Grid Portal Toolkit. https://gridport.npaci.edu/.

[7] D. B. Haidvogel, H. G. Arango, K. Hedstrom, A. Beckmannand, P. Malanotte-Rizzoli, and A. F. Shchepetkin. Model evaluation experiments in the North Atlantic Basin: Simulations in nonlinear terrain-following coordinates. *Dyn. Atmos. Oceans*, 32:239–281, 2000.

[8] Cay S. Horstmann and Gary Cornell. *Core Java 2: Volume I Fundamentals*. Sun Microsystems Press, Dec 2001.

[9] Java 2 Platform, Standard Edition v1.4.1 overview. http://java.sun.com/j2se/1.4.1/index.html.

[10] Summary of tools for the JavaTM 2 Platform security. http://java.sun.com/j2se/1.4.1/docs/guide/security/SecurityToolsSummary%.html.

[11] Trail: Creating a GUI with JFC/Swing. http://java.sun.com/docs/books/tutorial/uiswing/start/index.html.

[12] JDOM: Java Document Object Model. `http://www.jdom.org`.

[13] JNI – Java(TM) Native Interface. `http://java.sun.com/products/jdk/1.2/docs/guide/jni/`.

[14] Unidata NetCDF. `http://www.unidata.ucar.edu/packages/netcdf/`.

[15] N. M. Patrikalakis, S. L. Abrams, J. G. Bellingham, W. Cho, K. P. Mihanetzis, A. R. Robinson, H. Schmidt, and P. C. H. Wariyapola. The digital ocean. In *Proceedings of Computer Graphics International, GCI '2000*, pages 45–53, Geneva, Switzerland, June 2000. IEEE Computer Society Press. Los Alamitos, CA: IEEE, 2000.

[16] Nicholas M. Patrikalakis. Poseidon: A distributed information system for ocean processes. `http://czms.mit.edu/poseidon/`.

[17] O. F. Rana, M. Li, D. W. Walker, and M. Shields. An XML based component model for generating scientific applications and performing large scale simulations in a meta-computing environment. In *Proceedings of the First International Symposium on Generative and Component-Based Software Engineering*, Erfurt, Germany, September 1999.

[18] A. R. Robinson. Harvard Ocean Prediction System (HOPS). `http://oceans.deas.harvard.edu/HOPS/HOPS.html`.

[19] A. R. Robinson. Forecasting and simulating coastal ocean processes and variabilities with the Harvard Ocean Prediction System. In C.N.K. Mooers, editor, *Coastal Ocean Prediction*, AGU Coastal and Estuarine Studies Series, pages 77–100. American Geophysical Union, 1999.

[20] J. Roy and A. Ramanujan. XML schema language: Taking XML to the next level. *IT Professional*, 3(2):37–40, mar 2001.

[21] Simple API for XML (SAX). `http://www.saxproject.org/`.

[22] M. S. Shields, O. F. Rana, D. W. Walker, and M. Li. A Java/CORBA based visual program composition environment for PSEs. *Concurrency: Practice and Experience*, 12:687–704, 2000.

[23] H. M. Sneed. Wrapping legacy COBOL programs behind an XML-interface. In *Proceedings of the Eighth Working Conference on Reverse Engineering*, pages 189–197, 2001.

[24] A. Terekhov and C. Verhoef. The realities of language conversions. *IEEE Software*, pages 111–124, November 2000.

[25] D. W. Walker, M. Li, and O. F. Rana. An XML-based component model for wrapping legacy codes as Java/CORBA components. In *Proceedings of the Fourth International Conference on High Performance Computing in the Asia-Pacific Region*, pages 507–512, Beijing, People's Republic of China, May 2000. IEEE Computer Society Press.

[26] D. W. Walker, M. Li, O. F. Rana, M. S. Shields, and Y. Huang. The software architecture of a distributed problem-solving environment. *Concurrency: Practice and Experience*, 12(15):1455–1480, December 2000.

[27] Web services made easier: The Java APIs and architectures for XML technical white paper. `http://java.sun.com/xml/webservices.pdf`.

[28] eXtensible Markup Language. `http://www.w3.org/XML`.

[29] XML Schema Specification. `http://www.w3.org/XML/Schema`.

[30] Zhitao Yu. Harvard Ocean Prediction System (HOPS) USER'S GUIDE. October 2001.

Appendix A

Program Schema – *program.xsd*

```
<?xml version="1.0"?>

<!-- program.xsd SCHEMA        -->

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

   <!-- ********** PROGRAM ********** -->
   <xs:element name="program" type="programItem" />


   <!-- ********** DEFINITION OF program ELEMENT ********** -->
   <xs:complexType name="programItem">
      <xs:all>
         <xs:element name="name"         type="xs:string"       />
         <xs:element name="info"         type="xs:string"       />
         <xs:element name="architecture" type="archItem"        />
         <xs:element name="shell"        type="shellItem"       />
         <xs:element name="makefile"     type="makefileItem"
                                         minOccurs="0"          />
         <xs:element name="binaries"     type="binaryList"      />
         <xs:element name="constraints"  type="constraintsList"
                                         minOccurs="0"          />
      </xs:all>
   </xs:complexType>


   <!-- ********** DEFINITION OF architecture ELEMENT ********** -->
   <xs:complexType name="archItem">
      <xs:sequence>
         <xs:element name="choices"  type="archChoiceItem" />
         <xs:element name="selected" type="archTypes"  />
      </xs:sequence>
   </xs:complexType>


   <!-- ********** DEFINITION OF choices ELEMENT ********** -->
   <xs:complexType name="archChoiceItem">
      <xs:sequence>
         <xs:element name="choice"  type="archTypes"
                                    maxOccurs="unbounded" />
      </xs:sequence>
   </xs:complexType>


   <!-- ********** DEFINITION OF architecture TYPES ********** -->
   <xs:simpleType name="archTypes">
   <xs:restriction base="xs:string">
      <xs:enumeration value="alpha"  />
      <xs:enumeration value="cray"   />
      <xs:enumeration value="iris"   />
      <xs:enumeration value="linux"  />
      <xs:enumeration value="rs6000" />
      <xs:enumeration value="sun3"   />
      <xs:enumeration value="sun4"   />
      <xs:enumeration value="sun5"   />
   </xs:restriction>
</xs:simpleType>


<!-- ********** DEFINITION OF shell ELEMENT ********** -->
<xs:complexType name="shellItem">
   <xs:sequence>
      <xs:element name="choices"  type="shellChoiceItem" />
      <xs:element name="selected" type="shellTypes"  />
   </xs:sequence>
</xs:complexType>


<!-- ********** DEFINITION OF choices ELEMENT ********** -->
<xs:complexType name="shellChoiceItem">
   <xs:sequence>
      <xs:element name="choice"  type="shellTypes"
                                 maxOccurs="unbounded" />
   </xs:sequence>
</xs:complexType>


<!-- ********** DEFINITION OF shell TYPES ********** -->
<xs:simpleType name="shellTypes">
   <xs:restriction base="xs:string">
      <xs:enumeration value="sh"   />
      <xs:enumeration value="bash" />
      <xs:enumeration value="csh"  />
      <xs:enumeration value="tcsh" />
      <xs:enumeration value="ksh"  />
   </xs:restriction>
</xs:simpleType>


<!-- ********** DEFINITION OF makefile ELEMENT ********** -->
<xs:complexType name="makefileItem">
   <xs:sequence>
      <xs:element name="info"    type="xs:string"               />
      <xs:element name="type"    type="makefileTypes"           />
      <xs:element name="path"    type="xs:string"               />
      <xs:element name="xmlDesc" type="xs:string" minOccurs="0" />
   </xs:sequence>
</xs:complexType>


<!-- ********** DEFINITION OF makefile TYPES ********** -->
<xs:simpleType name="makefileTypes">
   <xs:restriction base="xs:string">
      <xs:enumeration value="GNUmake" />
      <xs:enumeration value="BSDmake" />
   </xs:restriction>
</xs:simpleType>


<!-- ********** DEFINITION OF binaries ELEMENT ********** -->
<xs:complexType name="binaryList">
   <xs:sequence>
      <xs:element name="binary" type="binaryItem"
                                maxOccurs="unbounded" />
      </xs:sequence>
   </xs:complexType>

   <!-- ********** DEFINITION OF binary ELEMENT ********** -->
   <xs:complexType name="binaryItem">
      <xs:sequence>
         <xs:element name="info"    type="xs:string"               />
         <xs:element name="arch"    type="archTypes"               />
         <xs:element name="path"    type="xs:string"               />
         <xs:element name="xmlDesc" type="xs:string" minOccurs="0" />
      </xs:sequence>
   </xs:complexType>


   <!-- ********** DEFINITION OF constraints ELEMENT ********** -->
   <xs:complexType name="constraintsList">
      <xs:sequence>
         <xs:element name="constraint" type="constraintItem"
                                       maxOccurs="unbounded" />
      </xs:sequence>
   </xs:complexType>


   <!-- ********** DEFINITION OF constraint ELEMENT **********  -->
   <xs:complexType name="constraintItem">
      <xs:sequence>
         <xs:element name="name"    type="xs:string"               />
         <xs:element name="info"    type="xs:string"               />
         <xs:element name="path"    type="xs:string"               />
         <xs:element name="xmlDesc" type="xs:string" minOccurs="0" />
      </xs:sequence>
   </xs:complexType>

</xs:schema>
```

Appendix B

Makefile Schema – *makefile.xsd*

```
<?xml version="1.0"?>

<!-- makefile.xsd SCHEMA       -->

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <xs:include schemaLocation="program.xsd" />

   <!-- ********** MAKEFILE ********** -->
   <xs:element name="makefile" type="makefileType" />


   <!-- ********** DEFINITION OF makefile ELEMENT ********** -->
   <xs:complexType name="makefileType">
      <xs:sequence>
         <xs:element name="path"      type="xs:string"      />
         <xs:element name="info"      type="xs:string"      />
         <xs:element name="section"   type="sectionItem"
                                      maxOccurs="unbounded" />
         <xs:element name="conflicts" type="conflictsList"  />
      </xs:sequence>
   </xs:complexType>


   <!-- ********** DEFINITION OF section ELEMENT ********** -->
   <xs:complexType name="sectionItem">
      <xs:sequence>
         <xs:element name="info" type="xs:string" />
         <xs:choice>
            <xs:element name="includeFileChoice" type="includeChoices" />
            <xs:element name="includeFile"       type="xs:string"      />
            <xs:element name="preproc-objects"   type="ppObjectsList"  />
         </xs:choice>
      </xs:sequence>
   </xs:complexType>


   <!-- ********** DEFINITION OF includeFileChoice ELEMENT ********** -->
   <xs:complexType name="includeChoices">
      <xs:sequence>
         <xs:element name="choice" type="makefileChoiceItem"
                                   maxOccurs="unbounded" />
      </xs:sequence>
   </xs:complexType>


   <!-- ********** DEFINITION OF choice ELEMENT ********** -->
   <xs:complexType name="makefileChoiceItem">
      <xs:sequence>
         <xs:element name="architecture" type="archTypes" />
         <xs:element name="includeFile"  type="xs:string" />
      </xs:sequence>
   </xs:complexType>


   <!-- ********** DEFINITION OF preproc-objects ELEMENT ********** -->
   <xs:complexType name="ppObjectsList">
      <xs:sequence>
         <xs:element name="startText"   type="xs:string"      />
         <xs:element name="endText"     type="xs:string"      />
         <xs:element name="separator"   type="xs:string"      />
         <xs:element name="preproc-obj" type="ppObjItem"
                                        maxOccurs="unbounded" />
      </xs:sequence>
   </xs:complexType>


   <!-- ********** DEFINITION OF preproc-obj ELEMENT ********** -->
   <xs:complexType name="ppObjItem">
      <xs:all>
         <xs:element name="name"          type="xs:string"    />
         <xs:element name="info"          type="xs:string"    />
         <xs:element name="header"        type="xs:string"    />
         <xs:element name="value"         type="xs:string"
                                          minOccurs="0"       />
         <xs:element name="use"           type="xs:boolean"   />
         <xs:element name="requires"      type="requiresList"
                                          minOccurs="0"       />
      </xs:all>
   </xs:complexType>


   <!-- ********** DEFINITION OF requires ELEMENT ********** -->
   <xs:complexType name="requiresList">
      <xs:sequence>
         <xs:element name="item"   type="xs:string"
                     minOccurs="0" maxOccurs="unbounded" />
      </xs:sequence>
   </xs:complexType>


   <!-- ********** DEFINITION OF conflicts ELEMENT ********** -->
   <xs:complexType name="conflictsList">
      <xs:sequence>
         <xs:element name="conflict" type="conflictItem"
                     minOccurs="0"   maxOccurs="unbounded" />
      </xs:sequence>
   </xs:complexType>


   <!-- ********** DEFINITION OF conflict ELEMENT ********** -->
   <xs:complexType name="conflictItem">
      <xs:sequence>
         <xs:element name="item"     type="xs:string"
                                     maxOccurs="unbounded" />
      </xs:sequence>
   </xs:complexType>

</xs:schema>
```

Appendix C

Binary Schema – *binary.xsd*

```
<?xml version="1.0"?>

<!-- binary.xsd SCHEMA          -->

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <xs:include schemaLocation="parameters.xsd" />

   <!-- ********** BINARY ********** -->
   <xs:element name="binary" type="binaryItem" />


   <!-- ********** DEFINITION OF binary ELEMENT ********** -->
   <xs:complexType name="binaryItem">
      <xs:sequence>
         <xs:element name="name"      type="xs:string"  />
         <xs:element name="path"      type="xs:string"  />
         <xs:element name="info"      type="xs:string"  />
         <xs:element name="constants" type="constList"  />
         <xs:element name="files"     type="fileList"   />
         <xs:element name="cl-args"   type="cl-argList" />
         <xs:element name="stdin"     type="CLItem"     />
      </xs:sequence>
   </xs:complexType>


   <!-- ********** DEFINITION OF constList ELEMENT ********** -->
   <xs:complexType name="constList">
      <xs:sequence>
         <xs:element name="constant" type="constItem"
                     minOccurs="0"   maxOccurs="unbounded" />
      </xs:sequence>
   </xs:complexType>


   <!-- ********** DEFINITION OF constant ELEMENT ********** -->
   <xs:complexType name="constItem">
      <xs:sequence>
         <xs:element name="name"  type="xs:string" />
         <xs:element name="info"  type="xs:string" />
         <xs:element name="value" type="xs:string" />
         <xs:element name="type"  type="xs:string" />
      </xs:sequence>
   </xs:complexType>


   <!-- ********** DEFINITION OF fileList ELEMENT ********** -->
   <xs:complexType name="fileList">
      <xs:sequence>
         <xs:element name="file"   type="fileItem"
                     minOccurs="0" maxOccurs="unbounded" />
      </xs:sequence>
   </xs:complexType>


   <!-- ********** DEFINITION OF cl-argList ELEMENT ********** -->
   <xs:complexType name="cl-argList">
      <xs:sequence>
         <xs:element name="cl-arg" type="CLItem"
                     minOccurs="0" maxOccurs="unbounded" />
      </xs:sequence>
   </xs:complexType>


   <!-- ********** DEFINITION OF CLItem ELEMENT ********** -->
   <xs:complexType name="CLItem">
      <xs:sequence>
         <xs:element name="switch" type="xs:string"      />
         <xs:element name="use"    type="xs:boolean"     />
         <xs:element name="info"   type="xs:string"      />
         <xs:element name="var"    type="varItem"
                     minOccurs="0" maxOccurs="unbounded" />
      </xs:sequence>
   </xs:complexType>

</xs:schema>
```

Appendix D

Parameters Schema – *parameters.xsd*

```
<?xml version="1.0"?>

<!-- parameters.xsd SCHEMA      -->

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

   <!-- ********** PARAMFILE ********** -->
   <xs:element name="paramfile" type="paramItem" />


   <!-- ********** DEFINITION OF parameters ELEMENT ********** -->
   <xs:complexType name="paramItem">
      <xs:sequence>
         <xs:element name="path"      type="xs:string"      />
         <xs:element name="info"      type="xs:string"      />
         <xs:element name="startText" type="xs:string"      />
         <xs:element name="endText"   type="xs:string"      />
         <xs:element name="set"       type="setItem"
                     minOccurs="0"    maxOccurs="unbounded" />
      </xs:sequence>
   </xs:complexType>


   <!-- ********** DEFINITION OF set ELEMENT ********** -->
   <xs:complexType name="setItem">
      <xs:sequence>
         <xs:element name="order"     type="xs:integer"     />
         <xs:element name="info"      type="xs:string"      />
         <xs:element name="startText" type="xs:string"      />
         <xs:element name="endText"   type="xs:string"      />
         <xs:element name="separator" type="xs:string"      />
         <xs:element name="var"       type="varItem"
                                      maxOccurs="unbounded" />
      </xs:sequence>
   </xs:complexType>


   <!-- ********** DEFINITION OF var ELEMENT ********** -->
   <xs:complexType name="varItem">
      <xs:sequence>
         <xs:element name="name"   type="xs:string" />
         <xs:element name="header" type="xs:string"
                                   minOccurs="0"    />
         <xs:choice>
            <xs:element name="uneditable" type="uneditableItem"  />
            <xs:element name="string"     type="stringItem"      />
         <xs:element name="enumerated" type="enumeratedItem"  />
         <xs:element name="integer"    type="intItem"         />
         <xs:element name="long"       type="longItem"        />
         <xs:element name="float"      type="floatItem"       />
         <xs:element name="double"     type="doubleItem"      />
         <xs:element name="file"       type="fileItem"        />
      </xs:choice>
      <xs:element name="use"  type="xs:boolean" />
   </xs:sequence>
   <xs:attribute name="type" type="varTypes" />
</xs:complexType>


<!-- ********** DATA TYPE DEFINITIONS ********** -->
<xs:complexType name="uneditableItem">
   <xs:sequence>
      <xs:element name="value" type="xs:string" />
      <xs:element name="info"  type="xs:string" />
   </xs:sequence>
</xs:complexType>


<xs:complexType name="stringItem">
   <xs:sequence>
      <xs:element name="value"     type="xs:string"  />
      <xs:element name="info"      type="xs:string"  />
      <xs:element name="minLength" type="xs:integer"
                                   minOccurs="0"     />
      <xs:element name="maxLength" type="xs:integer"
                                   minOccurs="0"     />
   </xs:sequence>
</xs:complexType>


<xs:complexType name="enumeratedItem">
   <xs:sequence>
      <xs:element name="value"            type="xs:string" />
      <xs:element name="info"             type="xs:string" />
      <xs:element name="acceptableValues" type="xs:string" />
   </xs:sequence>
</xs:complexType>


<xs:complexType name="intItem">
   <xs:sequence>
      <xs:element name="value" type="xs:integer" />
      <xs:element name="info"  type="xs:string"  />
      <xs:element name="units" type="xs:string"  />
      <xs:element name="range" type="xs:string"
                               minOccurs="0"     />
   </xs:sequence>
</xs:complexType>


<xs:complexType name="longItem">
   <xs:sequence>
      <xs:element name="value" type="xs:long"   />
      <xs:element name="info"  type="xs:string" />
      <xs:element name="units" type="xs:string" />
      <xs:element name="range" type="xs:string"
                               minOccurs="0"    />
   </xs:sequence>
</xs:complexType>


<xs:complexType name="floatItem">
   <xs:sequence>
      <xs:element name="value" type="xs:float"  />
```

```
         <xs:element name="info"  type="xs:string" />
         <xs:element name="units" type="xs:string" />
         <xs:element name="range" type="xs:string"
                                  minOccurs="0"    />
      </xs:sequence>
   </xs:complexType>


   <xs:complexType name="doubleItem">
      <xs:sequence>
         <xs:element name="value" type="xs:double" />
         <xs:element name="info"  type="xs:string" />
         <xs:element name="units" type="xs:string" />
         <xs:element name="range" type="xs:string"
                                  minOccurs="0"    />
      </xs:sequence>
   </xs:complexType>


   <xs:complexType name="fileItem">
      <xs:sequence>
         <xs:element name="id"        type="xs:string"          />
         <xs:element name="path"      type="xs:string"          />
         <xs:element name="info"      type="xs:string"          />
         <xs:element name="xmlDesc"   type="xs:string"          />
         <xs:element name="maxLength" type="xs:positiveInteger"
                                      minOccurs="0"             />
      </xs:sequence>
      <xs:attribute name="type" type="fileTypes" use="required" />
   </xs:complexType>


   <!-- ********** DEFINITION OF fileTypes TYPE  ********** -->
   <xs:simpleType name="fileTypes">
      <xs:restriction base="xs:string">
         <xs:enumeration value="input"  />
         <xs:enumeration value="output" />
         <xs:enumeration value="inout"  />
         <xs:enumeration value="other"  />
      </xs:restriction>
   </xs:simpleType>


   <!-- ********** DEFINITION OF varTypes TYPE  ********** -->
   <xs:simpleType name="varTypes">
      <xs:restriction base="xs:string">
         <xs:enumeration value="uneditable" />
         <xs:enumeration value="string"     />
         <xs:enumeration value="enumerated" />
         <xs:enumeration value="file"       />
         <xs:enumeration value="integer"    />
         <xs:enumeration value="long"       />
         <xs:enumeration value="float"      />
         <xs:enumeration value="double"     />
      </xs:restriction>
   </xs:simpleType>

</xs:schema>
```