Legacy Computing Markup Language (LCML) and LEGEND - LEGacy Encapsulation for Network Distribution

by

Stephen Kurt Geiger

B.S., Naval Architecture and Marine Engineering (2001)
Webb Institute

Submitted to the Department of Ocean Engineering
in partial fulfillment of the requirements for the degree of

Master of Science in Ocean Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2004

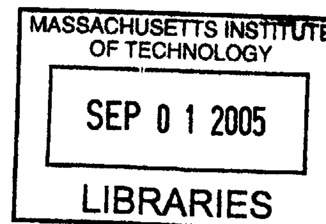
© Massachusetts Institute of Technology 2004. All rights reserved.

Author ....
Department of Ocean Engineering
May 26, 2004

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

Accepted by ....
Michael S. Triantafyllou
Chairman, Departmental Committee on Graduate Studies

BARKER



Legacy Computing Markup Language (LCML) and LEGEND - LEGacy Encapsulation for Network Distribution

by

Stephen Kurt Geiger

Submitted to the Department of Ocean Engineering
on May 26, 2004, in partial fulfillment of the
requirements for the degree of
Master of Science in Ocean Engineering

Abstract

The rapid increase of computing power and emergence of distributed computing technologies such as Grid computing create new opportunities for scientific computing. One of the challenges faced in harnessing the emerging computational power is how to effectively use traditional command-line driven "legacy" codes within a networked framework; and a related challenge is how to make the operation of such codes a more user-friendly process. In this work a specification for an XML-based Legacy Computing Markup Language (LCML) is developed. This language can be used to create a parametized encapsulation of command-line driven codes and their associated files. Such an encapsulation can then be viewed and edited with a program developed to process LCML descriptions. The program LEGEND (LEGacy Encapsulation for Network Distribution) is under development as a Java implementation of such a program. LEGEND demonstrates that a validating graphical user interface can automatically be generated from an LCML description. Some issues related to the encapsulation of legacy programs and use of LCML and LEGEND are discussed, as well as the possibilitites for the integration of these technologies with Sun Grid Engine (SGE) and Globus software.

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).

I'd like to express thanks to Dr. Constantinos Evangelinos for his assistance and insight throughout this project, and Professor Nicholas M. Patrikalakis for being my advisor and for his role in providing me the opportunity to study at MIT. I want to express thanks to Dr. Kwang Hee Ko and Harish Mukundan and appreciation as well to my friends and family.

Lastly, thanks to Jesus Christ by whom and for whom all has been created.

The earth is the Lord's and everything in it, the world and all who live in it; for he founded it upon the seas and established it upon the waters. - Psalm 24:1-2

Contents

1 Introduction **13**

2 Background **17**

2.1 Software Technologies 17

2.1.1 A Language for All Platforms - Java 17

2.1.2 Markup the World - XML 18

2.1.3 The Up and Coming - Grid Computing 20

2.2 Context and Related Work 21

2.2.1 The Poseidon Project 21

2.2.2 Examples of Related Work 23

3 The Legacy Computing Markup Language (LCML) **27**

3.1 The Role of LCML 27

3.2 LCML Overview 28

3.2.1 The `description` Element and its Children 29

3.2.2 The `descriptionChildren` Element and its Children 31

3.2.3 The `descriptionTarget` Element and its Children 32

3.2.4 The `descriptionContent` Element and its Children 33

3.2.5 The `set` Element and its Children 34

3.2.6 The `var` Element and its Children 36

3.2.7 The `block` Element and its Children 43

3.2.8 The `descriptionReplacements` Element and its Children . . . 46

3.2.9 The `descriptionConstraints` Element and its Children . . . 47

3.2.10 The Use of Dublin Core Metadata 52
3.2.11 The Use of References and Functions 55
3.2.12 The Use of Formatting Elements 58
3.2.13 The Display of Hidden Variables 58
3.2.14 The Resolution of a Variable's Value 58
3.2.15 The LCML Schema 59
3.3 Discussion of the LCML and its Development 60
3.3.1 The Inclusion of Dublin Core Metadata 60
3.3.2 Use of Elements vs. Use of Attributes 61
3.3.3 Use of One vs. Many Types of Descriptions 62
3.3.4 The Design of a Variable-Based Description 64
3.3.5 The Separation of Encapsulation and Processing 65
3.3.6 Multiplicity 66
3.3.7 Encapsulating Information 71
3.3.8 The Use of Validation 72
3.3.9 The Use of Constraints 74
3.3.10 Restricted Descriptions 75
3.3.11 The Use of Replacements 76
3.3.12 Creating and Running Scripts 77
3.3.13 Storing State - Import/Export Format 78
3.4 Example Descriptions 79
3.4.1 Hello World Example 79
3.4.2 Advanced Hello World Example 80
3.4.3 'cp' Example 83
3.4.4 SGE Example 86
3.5 Tools for Generating Descriptions 87
3.5.1 Use of an Existing XML Editor 88
3.5.2 Extension of a Text Editor 88
3.5.3 Use of LCML to Author Itself 90

4 **LEGEND (LEGacy Encapsulation for Network Distribution)** **91**
4.1 The Role of LEGEND 91
4.2 LEGEND Program Overview 92
4.2.1 Getting Started 92
4.2.2 The LEGEND Menu 94
4.2.3 The LCML Description Tree 97
4.2.4 The LCML Description Display Area 97
4.2.5 The Status Bar 101
4.3 Discussion of Design and Technical Considerations 101
4.3.1 User Interface Design and Implementation 101
4.3.2 The Use of Runtime.exec() 103
4.3.3 Extension of LEGEND 104
4.3.4 Validation, Constraints, and References 105
4.3.5 Additional Details 106

5 **Conclusions** **107**
5.1 Assessment of the Developed System 107
5.2 Recommendations for Future Work 108
5.2.1 Basic Tasks 108
5.2.2 Integration of Client/Server Capability 108
5.2.3 Integration with Globus 109
5.2.4 Development of an LCML API 109
5.2.5 Handling of Units 109
5.2.6 Enhanced Support of Mathematical Expressions 110
5.2.7 Formatted Input 110
5.2.8 User Interface Design and Usability of LCML 110
5.2.9 Display of Two-Dimensional Arrays 111
5.2.10 Automation in LCML/LEGEND 111
5.2.11 Automated Creation of LCML Descriptions 111

A **Legacy Computing Markup Language (LCML) Roadmap** **113**

B LCML Description Schema 119

List of Figures

3-1 Example of the Use of LCML in Encapsulation 28
3-2 Depiction of a Two-Dimensional Jagged Array 67
3-3 Prototype of LCML Authoring Extensions for Microsoft Word 89

4-1 Opening a description file. 92
4-2 Opening a remotely located description file. 93
4-3 Example of a set of opened LCML description files. 93
4-4 Example of opened LCML files with display areas labeled. 94
4-5 The Foldable Panel Display 97
4-6 The Sortable Table Display 98
4-7 The Display of Set Information via a Tooltip 99
4-8 The Display of Multiple Structures 101

Chapter 1

Introduction

Though space is often considered "The Final Frontier", the oceans that form vast portions of the earth's surface also remain in many ways a scientific frontier. There is much that is unknown about them. Aided by increases in available computing power and further motivated by the global scale of the nature of many of today's endeavors, efforts are being made to better understand this vast resource. A portion of these efforts fall into the domain of ocean prediction systems.

Ocean prediction systems are software systems that have a variety of goals and that can be helpful in dealing with several different ocean-related problems. Perhaps their most basic purpose is that they help the scientific community gain a better understanding of the ocean. This basic understanding is relevant to fields such as biology and oceanography. The understanding gained through ocean prediction systems can also be applied to fields that involve interaction with the ocean. Potential applications include pollution control, fisheries management, and naval operations.

The Harvard Ocean Prediction System (HOPS) [12] is an excellent example of an ocean prediction system. It is also an example of a "legacy" code. As used here, the term "legacy" does not imply the code itself is outdated; a legacy code can be under active development. Instead, the implication is that the approach used in developing the "legacy" software does not match current trends or the current state-of-the-art in software development. Such differences in approach can become problematic when one desires to effectively integrate these codes into a software environment that makes use

of modern techniques, and finds the "legacy" codes to be incompatible. In this work we have concerned ourselves with those legacy codes that are driven by a command-line and may employ file-based interactions.[1,2] Such codes are commonplace in the field of scientific computing.

A specific area of difficulty for the use of legacy codes is in the development of distributed computational systems with web-based access. In a best-case scenario, these systems are suited to handling the following concerns: effective use of heterogeneous computing resources, managing and concealing of computational complexities where possible, and ease of access. Particularly noteworthy is the ongoing development of "Grid computing" technologies [9, 10, 28]. Grid computing promises to facilitate better and more transparent use of diverse computing resources and will be discussed further in Chapter 2. With the incorporation of Grid computing, the best-case scenario we have mentioned here becomes more within reach; however, there remains a problem for legacy codes, in that they are not immediately suited to integration into such an environment.

A second issue is that legacy, "command-line driven" codes typically present some challenges for the user. An interface driven by command-line and file-based interactions can certainly be functional, and in some cases can be used effectively; however, such an interface will often lack much of the convenience that a graphical user interface (GUI) could provide. A relevant consideration here is that a well-designed GUI can also *substantially* lower the barriers associated with learning a new piece of software. The development of a GUI can make the operation of a program both easier to remember over time and easier to initially learn. This can be done by presenting visual cues and opportune information to the user. The development of a GUI also creates an opportunity to apply a measure of validation to user inputs. This can help to minimize erroneous, improper or invalid program runs.

[1] It is worth noting here that there are other forms of "legacy" codes. These include some that are driven by early "green-screen" user interfaces [2], or that run exclusively on legacy platforms. These are not so much our focus, and in many cases would present challenges altogether different from those we have dealt with here.

[2] One can also consider that the term "legacy", as applied to code and computing does not appear to have a specific, accepted, or singular definition. One might suggest that the codes we deal with in our work are better labeled "command-line driven" codes than "legacy".

In this work we consider how to deal with the issues of integration and interface of legacy, "command-line driven" codes: integration into a modern distributed computing environment and the automated generation of a relevant user interface. Our goal was not to develop a "stove-pipe" solution that was only applicable in a specific case or a small set of cases, but to develop a solution that would be generally applicable. Towards that end a two-tier solution is proposed.

First, legacy programs are encapsulated by describing them with the Legacy Computing Markup Language (LCML), a set of vocabulary and syntax for describing (and parametizing) legacy programs, their associated files, and the process used to build them. LCML has been developed as part of this thesis work and is written in eXtensible Markup Language (XML).

Second, user interfaces for descriptions written in LCML can be automatically generated by using an LCML processing program. Such a program has been developed in this thesis work under the name LEGEND (LEGacy Encapsulation for Network Distribution).[3] LEGEND provides a framework from which to generate files associated with a program and from which to write and run scripts for launching an encapsulated program. In its current state, LEGEND can be used to launch programs remotely by generating a script that makes use of software that allows for the remote execution of "command-line driven" programs.

The two-level encapsulation approach proposed here provides a general and effective way to deal with many "command-line driven" legacy codes. It allows for the possibility of an improved user experience and integration with modern software techniques such as Grid computing for a fraction of the effort that would typically be needed to develop a custom solution.

[3]LEGEND is currently under ongoing development and the full LCML specification is not yet supported.

Chapter 2

Background

This chapter discusses some of the underlying technologies that are relevant to this work, provides some information on the context within which this work was developed, and discusses some related work.

2.1 Software Technologies

There are three basic software technologies that are worthy of some discussion in relation to this work: Java, XML, and Grid Computing.

2.1.1 A Language for All Platforms - Java

Java is an object-oriented programming language developed by Sun Microsystems [14]. There are a number of characteristics about Java that make it well suited for use in this project. Java is:

- a widely used language - as such it has an active development community and there are a large number of resources available to learn Java, as well as a number of technologies that have been developed to work with Java.
- platform-independent - for this work it was desirable to develop a platform-independent solution. Java meets this requirement well.

- freely available - this fits well with making the developed software available as results of a publicly funded research project.
- a powerful modern programming language - Java does not impose undesirable limitations on the functionality we can effectively develop.
- capable of creating advanced graphical user interfaces (through the use of the Swing component library [13]) - this allows for a more advanced GUI display than we might get using a more typical web-based technology.
- well-suited to use over the web - Java has the power of applets which can be accessed through a web browser.

2.1.2 Markup the World - XML

The eXtensible Markup Language (XML) is a text format that provides a structured means for storage and exchange of data [8].

An XML Example

A sample XML document could appear as follows:

```
<root>
  <child anAttribute="attribute content">
    <grandchild>element content</grandchild>
  </child>
</root>
```

The above example demonstrates several of the basic concepts of XML which are relevant to this work.

- Bracketed items are tags that mark elements. Start tags are of the form '<elementName>' and end tags are of the form '</elementName>'. For example, the XML tags '`<root>`' and '`</root>`' mark the start and end of the `root` element.

- An element can have text content. In the above example, the text 'element content' is the text content of the `grandchild` element.

- An element can have other elements as content. In the above example, the `child` element is content of the `root` element.

- An element can have string-valued attributes associated with it. Attributes are embedded into an element's starting tag. In the above example, `anAttribute` is an attribute of the `child` element.

Markup Languages

While XML provides a general specification of a format for "marking-up" text, it does not provide a specific vocabulary or the specific data structure for doing so,[1] rather XML is "extensible". This makes it well suited as a master language for specifying a markup language. A markup language can be thought of as a dialect appropriate for describing the data relevant to a specific domain of information.

The best known example of a markup language (although not an XML-based one) is the Hypertext Markup Language (HTML). Examples of XML-based markup languages are the MathML and Scalable Vector Graphics (SVG) specifications.

To further illustrate the concept of a markup language consider the following example of an instance of a very simple, hypothetical, XML-based Point Markup Language:

```
<point>
  <x>0.0</x>
  <y>0.0</y>
</point>
```

This example demonstrates a manner in which the point with x and y coordinates each equal to zero could be represented in XML. If the format used in the example was

[1]In the previous example the `root` element could be named `node` or some other string, and it could have had attributes and/or any variation of elements and text content for its content.

defined as a standard specification for describing points, then it could be considered a "Point Markup Language".

Schemas for XML

An XML structure can be documented with a schema. An XML document is considered "schema-validated" if it has been shown to meet the format described in a schema.

If we consider the previous "Point Markup Language" example, our schema might serve to indicate the following:

- The root element of a point document is called `point`.
- A `point` element should have one `x` and one `y` element as its children.
- An `x` or `y` element should have a number as its only content.

While there are a several alternative schema languages including Document Type Definition (DTD), RELAX NG [25], and Document Structure Description (DSD) [5], in this work we have used the XML Schema language [30]. This language uses an XML-based format to describe schemas for XML documents. Rather than describe the details of using the XML Schema language here, the reader is pointed to [31] for a tutorial.

2.1.3 The Up and Coming - Grid Computing

Grid computing is currently a topic of much interest, especially as a tool for enabling advancements in scientific computing, but also potentially of considerable benefit to a variety of disciplines [9]. For our discussion, let us first consider the question, what is "the Grid"?

> "The Grid refers to an infrastructure that enables the integrated, collaborative use of high-end computers, networks, databases, and scientific instruments owned and managed by multiple organizations. Grid applications often involve large amounts of data and/or computing and often

> require secure resource sharing across organizational boundaries, and are thus not easily handled by todays Internet and Web infrastructures." [28]

Some of the potential benefits of Grid computing are as follows:

- Easy access to a variety of computing resources.
- The possibility of reclaiming otherwise unused computing cycles on idle computers. For large organizations with a sizable network of computers a significant amount of computational power can be reclaimed.
- Improved collaboration between organizations sharing data and computational resources.

There have been a number of Grid-related development projects, and several Grid-related technologies of interest include Globus [28], Sun Grid Engine [11], and Condor-G [4]. Far more could be written about the developing Grid technologies, but rather the reader is directed to the aforementioned references, as well as the Global Grid Forum [10], for more information.

2.2 Context and Related Work

2.2.1 The Poseidon Project

This work comprises a portion of the Poseidon research project, an endeavor of the MIT Ocean Engineering Design Laboratory and Acoustics Group in association with the Harvard University Interdisciplinary Ocean Science Group.

A basic description of the project is as follows:

> "Poseidon is ... a distributed computing based project that brings together advanced modeling, observation tools, and field and parameter estimation methods for oceanographic research. The project has three main goals:

1. to enable efficient interdisciplinary ocean forecasting, by coupling physical and biological oceanography with ocean acoustics in an operational distributed computing framework,
2. to introduce adaptive modeling and adaptive sampling of the ocean in the forecasting system, thereby creating a dynamic data-driven forecast,
3. and to initiate the concept of seamless access, analysis, and visualization of experimental and simulated forecast data, through a science-friendly Web interface that hides the complexity of the underlying distributed heterogeneous software and hardware resources. The aim is to allow the ocean scientist/forecaster to concentrate on the task at hand as opposed to the micro-management of the underlying forecasting mechanisms." [21]

Additional details on the project are available at the Poseidon project website [24] (as well as in [21]).

The Poseidon project makes use of the Harvard Ocean Prediction System (HOPS). For more information about HOPS, the reader is directed to the HOPS web site [12]. As mentioned previously, the HOPS system makes use of "command-line driven" legacy codes.[2] The current work is intended to allow for integration of HOPS into the project *without stipulating changes to the HOPS source code.*

The work presented in this thesis represents a continuation of work presented in the thesis "The Encapsulation of Legacy Binaries Using an XML-Based Approach with Applications in Ocean Forecasting" ([3]). In that work there is additional discussion of background issues such as XML and Java technologies and some additional comments on the ideas and approach that have been used in this project. A more concise discussion of the approach used can also be found in [7]. These references also provide some discussion of examples of similar work done elsewhere. We will now consider a few additional examples of related work.

[2]These form a number of smaller programs that are used to first setup and then work with an ocean model.

2.2.2 Examples of Related Work

Some examples of similar work are the following projects:

1. MAUI

A project developed at Sandia National Laboratories that has some similar ideas to our work is MAUI. From the MAUI project website [18]:

> "MAUI is a Java program for rapidly developing a graphical user interface (GUI) for an application based on a high-level XML description of the structure and parameters of the application. Custom actions can be plugged into MAUI to run the applications once the input parameters are entered into the GUI. The XML description can be easily modified to keep up with a developing project. Additionally, using MAUI can provide a consistent look and feel for related applications."

While MAUI shares a similar concept to our work, there are also some significant differences:

- MAUI outputs primarily to XML, and it appears that additional work must be done to have the XML converted to a desired text output.
- An XML description in MAUI can contain some specific details for how the GUI should be presented.[3]
- MAUI uses an object-oriented data representation. While object-oriented techniques are powerful, and this gives MAUI considerable flexibility in describing

[3]Encapuslating user interface details has pros and cons associated with it. In the author's estimation an encapsulation approach is better served by a cleaner break between program presentation and logic; and it is believed that this can be achieved by having an XML description conceptually model the points of interaction with the binary, and allowing the user interface to reflect the model inherent in the XML description. [It is worth noting that there are programs where the points of interaction are not easily modeled in XML. (A drawing program or a highly interactive program, e.g. full support of a command shell, come to mind as examples). Such a case might not be well suited to a separation of presentation and logic, but such cases are not really what we have in view for our development, and they introduce other complexities as well. They are not necessarily well-suited for encapsulation by our work or with MAUI.]

data, it is probably a fair assessment that this significantly complicates the writing of a description.

- The capabilities for validating data used in MAUI appear more limited than what is developed in this work.

2. Javamatic

Javamatic [22] is a tool that was developed to build web-based user interfaces to legacy command-line driven programs. The project dates back to 1997, and some details are not readily available; nor does it appear to be an active project.

It does provide an example of and details on how a client-server arrangement can effectively be used to enable web-based access to legacy codes.

In Javamatic the "encapsulation" of a legacy program was done at the level of Java classes. These could be written by a programmer or through the use of a graphical user interface that was developed for generating appropriate Java classes.

It is not clear how well the encapsulation capabilities were developed for handling difficult cases, or if any capability for validation was provided.

3. CAWOM

The CAWOM (Cal-Aggie Wrap-O-Matic) Project [29] is another effort that has developed tools for the wrapping of command-line driven programs. In the CAWOM approach programs are wrapped using code that is generated from the CAWOM-developed specifications. The generated code is compatible with the CORBA (Common Object Request Broker Architecture) standard. This approach allows the functionality of legacy codes to be incorporated programmatically into programs that make use of CORBA.

This approach provides a different result than our approach. In our approach once a program has been "encapsulated" (described in XML) a program that is capable of reading in LCML (such as LEGEND) can automatically generate a user interface. In the CAWOM approach the result of "wrapping" is that it is now easier to incorporate

the functionality of a legacy program into other CORBA capable programs. This approach is able to handle the wrapping of highly interactive[4] programs and is also well suited to cases where the legacy functionality needs to be mixed in an integrated way into a non-legacy program. However, with the CAWOM approach no capability for the automatic creation of a user interface is included.

4. PISE

PISE (Pasteur Institute Software Environment) [17], is a tool used to generate web interfaces for molecular biology programs. It makes use of XML encapsulation of command line programs and provides a web interface. It appears to be well-implemented and to receive significant use; however, it appears to be tuned to a specific application (in this case a large set of programs for molecular biology).

[4]CAWOM supports the parsing of program responses, a functionality we have not considered.

Chapter 3

The Legacy Computing Markup Language (LCML)

In this chapter we discuss the Legacy Computing Markup Language (LCML) which was developed as part of the work for this thesis. We will consider its function, overview the markup language itself, discuss the development process, and examine some examples of its use. We will then consider some tools that could provide assistance in generating LCML descriptions.

3.1 The Role of LCML

Legacy Computing Markup Language (LCML) is an XML-based markup language that is being developed to describe command-line driven binaries, the files associated with them, and the process used to compile such binaries. The concept is to provide a standardized syntax and vocabulary (expressed in XML) for the description of binaries, and their associated files and parameters. LCML can be used to describe scripts, input files, makefiles, and include files, and it provides a number of features to do so. With this approach of encapsulation of programs at the binary level it is not necessary to interact with the internals of a binary's operation. One can "communicate" with binaries through their command-line arguments and input files, and generally treat a binary's internal workings as a black box. The binary is dealt

with at its "edges" (as shown in Figure 3-1), and no modification or even knowledge of the binary's source code is required.[1]

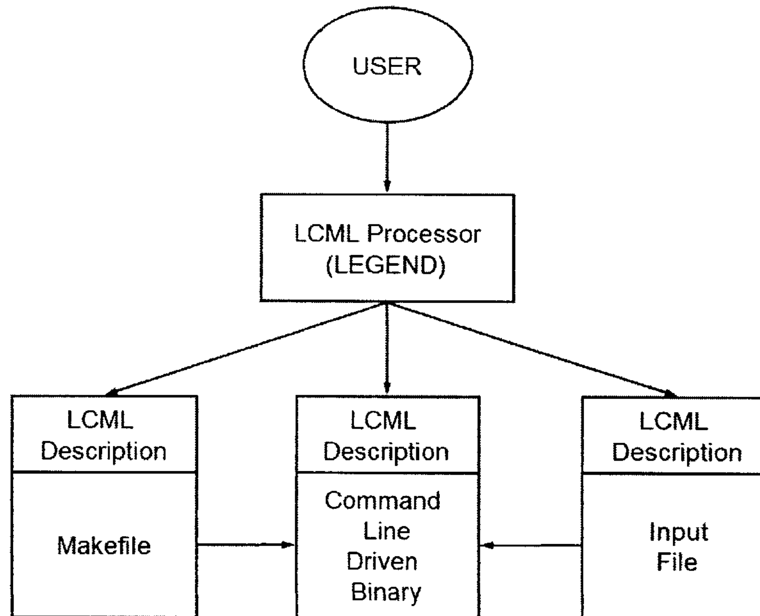


Figure 3-1: Example of the Use of LCML in Encapsulation

3.2 LCML Overview

Here we present an overview of the XML structure and element content used in writing a description with the LCML. For readability, the syntax used in this discussion is not in actual XML format; rather, it indicates the names of elements used and then represents the structure of the nesting of elements by level of indentation. Elements that are optional are placed in brackets ('[ ]'). An asterix ('*') placed by an element indicates that it can be used more than once. Unless marked or otherwise noted it is assumed that an element must be included once and only once.

In the discussion of some elements it will be noted that the element can make use

[1] This approach does not entirely preclude the consideration of the "settings" with which a binary was built and a method for handling this is discussed in Section 3.3.9.

of a "reference". A discussion of what "references" are and their use can be found in Section 3.2.11.

3.2.1 The `description` Element and its Children

Every LCML document should have a `description` element for its root element. The `description` element and the elements that are its children are shown here:

```
- description
  - descriptionName
  - descriptionInfo
  - [descriptionMetadata]
  - [descriptionChildren]
  - [descriptionTarget]
  - [descriptionContent]
  - [descriptionReplacements]
  - [descriptionConstraints]
```

The children of the `description` element have the following significance:

`descriptionName` - A string appropriate for naming and identifying this description. The name that is chosen should suggest the purpose or use of the description.

`descriptionInfo` - A more complete statement of the use and purpose of the description file. Any special information relevant to the description file should also be placed here.

`descriptionMetadata` - This optional element is used to provide information about the description (metadata) and can contain any of the fifteen standard Dublin Core metadata elements [6] as one of its children. The Dublin Core metadata elements will receive further description in Section 3.2.10.

`descriptionChildren` - This optional element is used to reference other LCML description files. This element and its children will be further described in Section 3.2.2.

`descriptionTarget` - This optional element is used to indicate that the description can be used to generate a file or a script. If it is not used no output should be created from the description. This element and its children will be further described in Section 3.2.3.

`descriptionContent` - This optional element contains the main content of the description. While this content is primarily used for the creation of output, it can also be used to describe information that is useful within a group of LCML description files.[2] This element and its children are further described in Section 3.2.4.

`descriptionReplacements` - This is an optional element. Its children can be used to describe substitutions to be made to portions of the text output that is created from the content section. This element and its children are described further in Section 3.2.8.

`descriptionConstraints` - This is an optional element. Its children can be used to describe constraints on the content of descriptions and on the content of any replacements. This element and its children are described further in Section 3.2.9.

[2]For example: an array index may be specified among the content, but not directly included in the output.

3.2.2 The descriptionChildren Element and its Children

The `descriptionChildren` element is used to indicate that a description has child descriptions.[3] If it is used, it should contain at least one `descriptionChild` element. The `descriptionChildren` element and its child element are shown here:

```
- [descriptionChildren]
  - descriptionChild*
```

The `descriptionChild` element is used to indicate an instance of a child LCML description. The `descriptionChild` element and its children are shown here:

```
- descriptionChild*
  - [location]
  - [absoluteLocation]
  - [description]
  - [association]
```

The children of the `descriptionChild` element have the following significance:

`location` - This optional element is used to indicate the path to the child description file. If this element is used, the path should be specified relative to the directory that the parent description file[4] is located in. A reference can be used here.[5] While the `location` element is optional, either the `location`, `absoluteLocation`, or `description` element should be used.

`absoluteLocation` - This optional element used to indicate the absolute path to the child description file. The path can be specified to a local file or as a uniform resource locator (URL). A reference can be used here. While the

[3] A 'child description' is a separate description file that is associated with its parent description. For example, a description that describes a program may have a description of the program's standard input file associated with it as its child.

[4] That is the description file that contains this element.

[5] See Section 3.2.11 for a discussion of references.

absoluteLocation element is optional, either the location, absoluteLocation, or description element should be used.

description - This optional element mirrors the root description element. It is used to directly define a child description as opposed to referencing an external description file. While the description element is optional, either the location, absoluteLocation, or description element should be used.

association - This optional element is used to indicate the relationship between a child description and a variable.[6,7] The association element is provided for information only.

3.2.3 The descriptionTarget Element and its Children

The descriptionTarget element is used to indicate that this description can produce file or script output. The descriptionTarget element and its children are shown here:

```
- [descriptionTarget]
  - [file]
  - [script]
```

The children of this descriptionTarget element have the following significance:

file - This optional element is used to indicate the path that the output of this description should be saved to if an output file is created from this description. A reference can be used here. While the file element is optional, either the file element or script element should be used.

[6] The "associated" variable should be of the 'file' or 'binary' variable-type.

[7] The "associated" variable should not be in a block of multiplicity (see Section 3.2.7); such an association would be amibiguous.

`script` - This optional element is used to indicate the information to necessary to run a script created from this description. While the `script` element is optional, either the `file` element or `script` element should be used.

The `script` element and its children are shown here:

```
- [script]
  - command
  - scriptname
```

The children of the `script` element have the following significance:

`command` - This element is used to describe a command that can be used to run the script that is output from the description.[8] A reference can be used here.

`scriptname` - This element indicates the filename (and possibly path) of the script to which the output text of the description should be written. The scriptname indicated here should also be included in the text of the `command` element.[9] A reference can be used here.

3.2.4 The `descriptionContent` Element and its Children

The `descriptionContent` is primarily a container for `set` elements; it should have at least one `set` element as a child and can have more. Optionally, the `startText`, `endText`, and `separator` elements can be used for additional control in formatting the output of the content. The `descriptionContent` element and the elements that are its children are shown here:

[8] Note that the exact specification of what this text should contain has some dependence on how the description will be processed (i.e. how the the script will be launched). For example, if the description is to be processed in a Java program that makes use of the "Runtime.exec()" method, then the command should be written in manner that can be properly interpreted by the "Runtime.exec()" method.

[9] For example, to run the command: 'sh tempscript' (where the intention is to use the 'sh' shell program to run the script named 'tempscript'), the text of the `command` element should be 'sh tempscript' and the text of the `scriptname` element should be 'tempscript'.

- [descriptionContent]
  - [startText]
  - [endText]
  - [separator]
  - set*

The children of the `descriptionContent` element have the following significance:

`set` - This element is used to group the content and it is discussed in detail in Section 3.2.5.

`startText` - This optional element is used when creating a script or file output from the content. Whatever text is contained within the `startText` element is included in the output text prior to the output from any sets.

`endText` - This optional element is used when creating a script or file output from the content. Whatever text is contained within the `endText` element is included in the output text after the output from all the sets.

`separator` - This optional element is used when creating a script or file output from the content. Whatever text is contained within the `separator` element is inserted in the output text in between the output created from sets.

Next, let us consider the `set` element and its children.

3.2.5 The set Element and its Children

The `set` element acts as a container for any number of, order of, and combination of `var` and `block` elements. The methodology for grouping of variables, and blocks into sets should be based on semantic reasons. A `set` should therefore represent a

group of related content.[10] The `set` element and the elements that are its children are shown here:

```
- set*
  - setName
  - setInfo
  - [startText]
  - [endText]
  - [separator]
  - [var]*
  - [block]*
```

The children of the `set` element have the following significance:

`setName` - A string appropriate for naming and identifying this set. The name that is chosen should describe the the purpose of the set, or otherwise identify the reason for the grouping of the set's contents into a set.

`setInfo` - A more complete description of the use, purpose, and/or grouping of the set. Special information relevant to the set can also be placed here.

`startText` - This optional element is used when creating a script or file output from the content. Whatever text is contained within the `startText` element is included in the output text prior to the output from the variables and blocks within the set.

`endText` - This optional element is used when creating a script or file output from the content. Whatever text is contained within the `endText` element is included in the output text after the output from the variables and blocks within the set.

[10] If all parameters in the content are related to the same function then it is reasonable to only have one `set`.

`separator` - This optional element is used when creating a script or file output from the content. Whatever text is contained within the `separator` element is inserted in the output text in between the output created from the variables and blocks within the set.

`var` - This element is used to represent a variable, which is the base unit of much of the content of the description. It is a complicated element and it is described in detail in Section 3.2.6. Any number of `var` elements can be placed within a set (and they can be intermingled with `block` elements), but a set should contain at least one `var` or `block` element.

`block` - This element is used to handle a case where a variable has a multiplicity of values. A basic example of this is an array. Its implementation is done with flexibility to allow for nested data structures and to support one-dimensional, two-dimensional, and in the most complex scenario: n-dimensional, multi-typed, jagged arrays where not all values are part of the output. The details of its implementation are discussed in Section 3.2.7. Any number of `block` elements can be placed within a set (and they can intermingled with `var` elements), but a set should contain at least one `var` or `block` element.

Next, let us consider the `var` element and its children.

3.2.6 The `var` Element and its Children

The `var` element is used to represent a variable, the basic unit of content in an LCML description. The primary use of a variable is to store (and provide a means to edit) the value of a parameter that is to be written to a script or file; however, a variable can also be used for information relevant to working with a program and its files. (In other words, the value of a variable may not always appear in the file or script that is created from the output of a description).

While all `var` elements have a number of child elements in common, there are also

a number of “type-dependent elements” that are only relevant as children for certain types of variables. The var element and its children are shown here:

- [var]*
  - name
  - info
  - type
  - value
  - use
  - hidden
  - [header]
  - [trailer]
  - [enumeration]
  - [uneditable]
  - [aliases]
  - [precision]
  - [range]
  - [units]
  - [minLength]
  - [maxLength]
  - [multiLine]
  - [fileType]
  - [architecture]

The children of the var elements have the following significance:

name - A string appropriate for naming and identifying a variable. *Each variable in a description file should have a unique name.*

info - A string that provides information about a variable's significance and purpose, and any details relevant to its proper use.

`type` - A string that indicates the type of variable. The possible variable types are:

- 'numeric' - used to represent numeric data (makes use of the `precision`, `range`, and `units` type-dependent elements).
- 'string' - a general representation for string data (makes use of the `minLength`, `maxLength`, and `multiLine` type-dependent elements).
- 'file' - represents the path to and information about a file (makes use of the `maxLength` and `fileType` type-dependent elements).
- 'binary' - used to include the path to an compiled program[11] (makes use of the `architecture` type-dependent element).

`value` - The text of this element typically represents content that can be written as output or content that provides information relevant to the use of a description. A reference can be used here.[12]

`use` - This element should contain the text 'true' or 'false' or a reference that resolves to 'true' or 'false'. This element describes whether this variable's value should be included in the text of a file or script output that is created from this description. If a variable is not "used" (i.e. the text of the `use` element equals 'false') then no output should be created from it. If a variable is not "used" it can still have significance to the description it is just not included as part of the output text.

`hidden` - This element should contain the text 'true' or 'false' or a reference that resolves to 'true' or 'false'. This element describes whether a variable should be

[11] i.e. a "binary".

[12] We should note one special case. If this variable has a 'numeric' type and the text of this element begins with an equal sign ('=') then numeric validation is not performed on this value and the beginning equal sign is not written to output. This allows an expression to be written in place of a numeric value, as some command-line driven programs can make use of expressions in place of a value.

made visible to a user working with this LCML description. If a variable is "hidden" (that is the text of the `hidden` element equals 'true') its value is still included in output that is created from the description (and also can have other significance to the description), but it is generally not made visible to the user for display or editing.

`header` - This optional element is used when output is created from the variable. The text of the `header` element is written to the output text just prior to the text that results from the variable's value. If a variable's value is not "used" than the `header`'s text is not included in the output either. If the `header` element is omitted there is no header text for this variable.

`trailer` - This optional element is used when output is created from the variable. The text of the `trailer` element is written to the output text right after the text that results from the variable's value. If a variable's value is not "used" than the `trailer`'s text is not included in the output either. If the `trailer` element is omitted there is no trailer text for this variable.

`enumeration` - This optional element contains an expression used to enumerate a list of choices of valid content for the value element. The semi-colon character ';' is used to separate the choices in the list. For example: the enumeration text 'red;green;blue' would limit a variable's value to be either 'red', 'green', or 'blue'. Since the semi-colon ';' is used as the list separator, it cannot be used within the text of an item in the list.

`uneditable` - This optional element can have one of the following string values: 'value', 'use', or 'value;use'. The values indicate respectively, that the `value` element should be uneditable by an LCML description user, that the `use` element should be uneditable by an LCML description user, or that both the `value` and the `use` element should be uneditable by an LCML description user.

`aliases` - This optional element is used to provide a replacement functionality. It is shown here with its descendants:

- [aliases]
  - alias*
    - aliasKey
    - aliasOutput

The descendants of the `aliases` element have the following significance:

`alias` - This element represents an instance of an alias.

`aliasKey` - If the text of this element is found within this variable's value, the text of the `aliasOutput` element is substituted in its place. A reference can be used here.

`aliasOutput` - This text is substituted in place of the text of the aliasKey, if the text of the `aliasKey` element is found within this variable's value. A reference can be used here.

The following elements are only applicable for certain types of variables (as indicated by the content of the `type` element):

`precision` - This optional element is used to indicate the intended precision of a 'numeric' variable's value and should have one of the following string values: 'integer', 'float', 'long', or 'double'.[13] If a variable's type is 'numeric' and the `precision` element is not used, no assumptions should be made about the variable's intended precision.

[13]Note that there is a limitation here in that the specification of what exactly is an 'integer', 'float', 'long', and 'double' types is a language and programming model (e.g. 32 vs. 64 bit) dependent concept. There is however, broad argeement for 'float' and 'double' for languages that conform to the IEEE754 standard.

`range` - This optional element can contain a string that indicates the valid range for a 'numeric' variable. The basic syntax of the range notation string is best described by an example: '(0,1]'. This example indicates that the numeric value must be greater than zero and less than or equal to one.

The following rules apply to the range expression:

- It should start with an opening bracket or parenthesis ('[' or '(') and end with a closing bracket or parenthesis (']' or ')').
- Inside the brackets or parenthesis should be the minimum numeric value, a comma (','), and then the maximum numeric value.
- The strings '-INFINITY' and 'INFINITY' can be used to represent negative and positive infinity, respectively.
- The use of a bracket ('[' or ']') indicates the range is inclusive (greater than or equal to/less than or equal to) versus the use of a parenthesis('(' or ')') indicates the range is exclusive (greater than/less than).
- More than one range expression can be included in the range element, and multiple ranges should be separated by semi-colons (';').

The following is an example of a range expression that demonstrates these rules: '(0,1];[200,INFINITY)'. This range expression indicates that the numeric value of the variable should be greater than zero and less than or equal to one, *or* greater than 200 (and less than ∞).

`units` - An optional element that can contain a string that indicates the units associated with a 'numeric' variable's value. Currently this element is only used for informational purposes.

`minLength` - An optional element that contains a non-negative integer indicating the minimum number of characters of text of the value of a 'string' variable.

`maxLength` - An optional element that contains a non-negative integer indicating the maximum number of characters of the text of the value of a 'string' or 'file' variable.

`multiLine` - An optional element. If the value of its text is 'true' it indicates that a 'string' variable can contain more than a single line of text. If this element is not used it is assumed that a string variable's value represents a single line of text.

`fileType` - An optional element for 'file' variables. If used, its text can have one of the following values 'input', 'output', 'input;output', 'includeFile', or 'template'.

The first three values indicate respectively that a file is:

- used as an input file,
- used as an output file,
- used for both input and output.

If the file type is 'includeFile':

- the output of the variable is the content of the file whose path is represented by the value element (instead of the path).

The 'template' file type is:

- similar to the 'includeFile' type, but it assumes the path represented in the value element is specified relative to the directory where the description file that contains this element is located.

`architecture` - An optional informational element for 'binary' variables. It is used to indicate the architecture (platform) that a binary was built to run on.

3.2.7 The block Element and its Children

The block element is used to represent repeated data (for example, an array of data). The block element and its two children are shown here:

```
- [block]*
  - structure
  - data
```

The structure and data elements have the following significance:

structure - This element is used to describe the structure of the repeated data and is used to specify the content (variables and other structures) used in the structure.

data - This element is used to hold the values of the repeated data.

Now, we will consider the structure and data elements.

The structure element and its children are shown here:

```
- structure
  - structureName
  - structureInfo
  - [startText]
  - [endText]
  - [separator]
  - numOccurs
  - [var]*
  - [structure]*
```

The children of the structure element are described here:

`structureName` - A string appropriate for naming and identifying this structure.

`structureInfo` - A string describing the use of or information about the structure.

`startText` - This optional element is used when creating a script or file output from the content. Whatever text is contained within the `startText` element is included in the output text prior to the output from the structure.

`endText` - This optional element is used when creating a script or file output from the content. Whatever text is contained within the `endText` element is included in the output text after the output from the structure.

`separator` - This optional element is used when creating a script or file output from the content. Whatever text is contained within the `separator` element is inserted in the output text in between the output created from each repeat of the structure.

`numOccurs` - A numeric value indicating the number of times a structure is repeated. A reference can be used here.

`var` - The `var` element uses the same description as presented in Section 3.2.6; however, the child `value` element becomes a default value and the settings for "use" and "hidden" become applicable to all repeats. (The repeated variable's values are stored in the `data` element that corresponds to this stucture). Any number of `var` elements can be placed within a stucture (and they can be intermingled with child `structure` elements), but a structure should contain at least one `var` or child `structure` element.

`structure` - The `structure` element is recursive. It can contain a child element with different content. *This allows a block to represent a nested (n-dimensional) data structure.* Any number of child `structure` elements can be placed within a

structure (and they can intermingled with var elements), but a structure should contain at least one var or child structure element.

The data element and its children and grandchildren are shown here:

```
- data
  - [dataInstance]*
    - [value]*
    - [data]*
```

The children and grandchildren of the data element are described here:

dataInstance - The dataInstance element represents one repeat of the data for a structure. The number of dataInstance elements in a data section should match the value of the numOccurs element for the corresponding structure.

value - This element is used to represent a repeated value of a variable. The number and order of value and data elements should match the number and order of var and structure elements for the corresponding structure. References can be used here.

data - Just as structure elements can be recursive, the data element can be recursive, and each data element represents the data for a corresponding structure element. The number and order of value and data elements should match the number and order of var and structure elements for the corresponding structure.[14]

[14] The following is meant to illustrate the relationships between structure and data elements:

The data child element of a block element corresponds to the structure child element of the same block. If this first level structure element contains two child structures the first level data element would have in each of its dataInstance elements two data elements (corresponding to the first and second child structure).

3.2.8 The descriptionReplacements Element and its Children

The descriptionReplacements element allows for the text output created from the descriptionContent to serve as a "template" or "boilerplate" for text content. Specific text created from the replacementContent elements can be substituted into the "template" at desired locations. This can effectively handle a scenario where there is a large amount of text to be created by a description, but much of that text is invariant (for example, the description of a complex makefile). If the descriptionReplacements element is used, it should contain at least one replacement element. The descriptionReplacements element and its children and grandchildren are shown here:

```
- [descriptionReplacements]
  - replacement*
    - replacementKey
    - replacementContent
```

These elements have the following significance:

`replacement` - This element represents a section of replacement. There can be one or more than sections of replacement. Each section has its own `replacementKey` and `replacementContent`.

`replacementKey` - This element should contain a string value. When the output of the `descriptionContent` is created any occurences of this string will be replaced with the output that results from the content of the corresponding `replacementContent` element. A reference can be used here.

`replacementContent` - This element essentially mirrors the `descriptionContent` element described in Section 3.2.4. The only difference is that the output created by this content is used when the `replacementKey` text is found in the output of the `descriptionContent`. This output is substituted in place of the `replacementKey`

text at any place that the replacementKey text occurs.

3.2.9 The descriptionConstraints Element and its Children

The descriptionConstraints element is used to describe specific criteria for variable values and/or relationships between variables. Specific tests are called "constraints", and are expressed either in terms of "requirements" or "conflicts".[15] If the descriptionConstraints element is used, it should contain at least one requirement or conflict element. The descriptionConstraints element and its children are shown here:

```
- [descriptionConstraints]
  - [requirement]*
  - [conflict]*
```

The requirement and conflict elements have the following significance:

requirement - The requirement element is used to describe a case where if specific criteria for variable values and/or relationships between variables are true it is expected that certain other specific criteria and relationships are also true. A requirement is considered violated if its "condition" returns a 'true' result and what it "requires" returns a 'false' result.

conflict - The conflict element is used to describe a case where a certain set of specific criteria for variable values and/or relationships between variables are expected to not all be true. A conflict is violated if everything specified in it returns a 'true' result.

Now, we will consider the requirement and conflict elements in more detail.

[15] Anything that can be described using "requirements" should also be describable using "conflicts" and vice versa; however, in some cases one or the other may prove far more convenient.

The `requirement` element and it children and grandchildren are shown here:

```
- [requirement]*
  - explanation
  - condition
    - [test]*
    - [group]*
  - requires
    - [test]*
    - [group]*
```

The children and grandchildren of the `requirement` element have the following significance:

`explanation` - A textual explanation of the constraint. (It should be written to be easily understandable).

`condition` - The `condition` element can contain `test` and `group` elements. A condition returns a 'true' result if all tests and groups within it return a true result.

`requires` - The `requires` element can contain `test` and `group` elements. A requires element returns a 'true' result if all tests and groups within it return a true result (and otherwise returns a false result).

`test` - The `test` element is the basic unit of the constraint specification and is described in detail below. It can be evaluted to return 'true' or 'false'. Any number of `test` elements can be placed within a `condition` or `requires` element (and they can be intermingled with `group` elements), but both the `condition` and `requires` elements should contain at least one `test` or `group` element.

`group` - The `group` element is used to group a number of tests and is described in

detail below. It can be evaluated to return 'true' or 'false'. Any number of `group` elements can be placed within a `condition` or `requires` element (and they can be intermingled with `test` elements), but both a `condition` and a `requires` element should contain at least one `test` or `group` element.

The `conflict` element and its children are shown here:

- conflict
  - explanation
  - [test]*
  - [group]*

The children of the `conflict` element have the following significance:

`explanation` - A textual explanation of the constraint. (It should be written to be easily understandable).

`test` - The `test` element is the basic unit of the constraint specification and is described in detail below. It can be evaluted to return 'true' or 'false'. Any number of `test` elements can be placed within a conflict (and they can be intermingled with `group` elements), but a conflict should contain at least one `test` or `group` element.

`group` - The `group` element is used to group a number of tests and is described in detail below. It can be evaluated to return 'true' or 'false'. Any number of `group` elements can be placed within a conflict (and they can be intermingled with `test` elements), but a conflict should contain at least one `test` or `group` element.

The `test` element and its children are shown here:

- [test]*
  - item
  - relation

- item

The children of the test element have the following significance:

item - A string, a numeric value, or a reference[16] can be used here.

relation - This element should contain one of the following strings describing the expected relationship between the item elements:

- 'SAME' - true if two items contain strings that are identical
- 'DIFF' - true if two items contain strings that are not identical
- 'EQ' - true if two items contain values that are numerically equal
- 'NEQ' - true if two items contain values that are not numerically equal
- 'LT' - true if the value of the first item is numerically less than the value of the second item.
- 'LEQ' - true if the value of the first item is numerically less than or equal to the value of the second item.
- 'GT' - true if the value of the first item is numerically greater than the value of the second item.
- 'GEQ' - true if the value of the first item is numerically greater than or equal to the value of the second item.

item - Usage is the same as instance of item above.

A test returns a 'true' result if the relationship described by the relation element matches the actual relationship between the two item elements, otherwise it returns

[16]See Section 3.2.11 for an explanation of the use of references from this element as they represent a special case.

a 'false' result.

The `group` element and its children are shown here:

- [group]*
  - [operator]
  - [test]*
  - [group]*

The children of the `group` item have the following significance:

`operator` - This optional element should have one of the following string values: 'AND', 'OR, or "XOR'. If the operator element is not included the default operator is 'AND'.

`test` - See description above. Any number of `test` elements can be placed within a group (and they can be intermingled with `group` child elements), but a group should contain a total of at least *two* `test` and/or `group` child elements.

`group` - The `group` element can be used recursively, a group can contain another group. Any number of `group` child elements can be placed within a group (and they can be intermingled with `test` elements), but a group should contain a total of at least *two* `test` and/or `group` child elements.

The significance of the `operator` element follows the normal rules for logical operators and a group returns a 'true' result if:

- the operator equals 'AND' and all tests and groups contained in it return 'true' results.
- the operator equals 'OR' and any test or group contained in it returns a 'true' result.

- the operator equals 'XOR' and exactly one of the tests and groups contained in it returns a 'true' result.

3.2.10 The Use of Dublin Core Metadata

The term *metadata* can be loosely defined as "data about data". The Dublin Core Metadata Initiative (DCMI) is an organization that is commited to the development of metadata standards. Among the work done by the Dublin Core Metada Initiative has been the creation of a core set of metadata for the documentation of electronic documents. The Dublin Core metadata consists of fifteen parameters and can be expressed as elements in XML. As mentioned previously, these elements can optionally be included as documentation in an LCML description.

The Dublin Core elements are described here, using descriptions from the DCMI website [6]. Note that: any of these elements can also include the `xml:lang` attribute in order to indicate the langauge used in their encoding.

`dc:title` - A name given to the resource. Typically, Title will be a name by which the resource is formally known.

`dc:creator` - An entity primarily responsible for making the content of the resource. Examples of Creator include a person, an organization, or a service. Typically, the name of a Creator should be used to indicate the entity.

`dc:subject` - Subject and Keywords. A topic of the content of the resource. Typically, Subject will be expressed as keywords, key phrases or classification codes that describe a topic of the resource. Recommended best practice is to select a value from a controlled vocabulary or formal classification scheme.

`dc:description` - An account of the content of the resource. Examples of Description include, but is not limited to: an abstract, table of contents, reference to a graphical representation of content or a free-text account of the content.

`dc:publisher` - An entity responsible for making the resource available. Examples of Publisher include a person, an organization, or a service. Typically, the name of a Publisher should be used to indicate the entity.

`dc:contributor` - An entity responsible for making contributions to the content of the resource. Examples of Contributor include a person, an organization, or a service. Typically, the name of a Contributor should be used to indicate the entity.

`dc:date` - A date of an event in the lifecycle of the resource. Typically, Date will be associated with the creation or availability of the resource. Recommended best practice for encoding the date value is defined in a profile of ISO 8601 and includes (among others) dates of the form YYYY-MM-DD.

`dc:type` - Resource Type. The nature or genre of the content of the resource. Type includes terms describing general categories, functions, genres, or aggregation levels for content. Recommended best practice is to select a value from a controlled vocabulary (for example, the DCMI Type Vocabulary). To describe the physical or digital manifestation of the resource, use the FORMAT element.

`dc:format` - The physical or digital manifestation of the resource. Typically, Format may include the media-type or dimensions of the resource. Format may be used to identify the software, hardware, or other equipment needed to display or operate the resource. Examples of dimensions include size and duration. Recommended best practice is to select a value from a controlled vocabulary (for example, the list of Internet Media Types defining computer media formats).

`dc:identifier` - Resource Identifier. An unambiguous reference to the resource within a given context. Recommended best practice is to identify the resource by means of a string or number conforming to a formal identification system. Formal

identification systems include but are not limited to the Uniform Resource Identifier (URI) (including the Uniform Resource Locator (URL)), the Digital Object Identifier (DOI) and the International Standard Book Number (ISBN).

`dc:source` - A Reference to a resource from which the present resource is derived. The present resource may be derived from the Source resource in whole or in part. Recommended best practice is to identify the referenced resource by means of a string or number conforming to a formal identification system.

`dc:language` - A language of the intellectual content of the resource. Recommended best practice is to use RFC 3066 which, in conjunction with ISO639), defines two- and three-letter primary language tags with optional subtags. Examples include "en" or "eng" for English, "akk" for Akkadian", and "en-GB" for English used in the United Kingdom.

`dc:relation` - A reference to a related resource. Recommended best practice is to identify the referenced resource by means of a string or number conforming to a formal identification system.

`dc:coverage` - The extent or scope of the content of the resource. Typically, Coverage will include spatial location (a place name or geographic coordinates), temporal period (a period label, date, or date range) or jurisdiction (such as a named administrative entity). Recommended best practice is to select a value from a controlled vocabulary (for example, the Thesaurus of Geographic Names) and to use, where appropriate, named places or time periods in preference to numeric identifiers such as sets of coordinates or date ranges.

`dc:rights` - Rights Management. Information about rights held in and over the resource. Typically, Rights will contain a rights management statement for the resource, or reference a service providing such information. Rights information often

encompasses Intellectual Property Rights (IPR), Copyright, and various Property Rights. If the Rights element is absent, no assumptions may be made about any rights held in or over the resource.

Upon reading through the descriptions of the Dublin Core elements one might question the relevance of some of these elements for the role of metadata for LCML descriptions. This was considered, but rather than attempting to carve a subset of relevant elements out of the Dublic Core elements, a descision was made to support all the core elements and to let the authors of LCML descriptions decide which of the elements to include as documentation for their own descriptions.

3.2.11 The Use of References and Functions

References

"References" are used to return a text string corresponding to the 'value' or 'use' setting of the referenced variable. The resultant text string is then used in place of the reference. There are a number of elements in LCML that references can be made from and these elements have been noted as they were described.

This method allows the flexibility for the content of some elements that are not variables (for example, a path associated with the `file` element of a `descriptionTarget` element) to depend on a variable. The referenced variable can then be displayed and edited by an LCML user in the same manner as any other variable. References can be used in a number of other ways as well, and can make the editing process be less labor-intensive for the user.

References themselves are also described with a text string. To reference the 'value' of a variable one of following two forms is used:

- REF(VariableName)
- REF(DescriptionName;VariableName)

To reference the 'use' setting of a variable one of following two forms is used:

- USEREF(VariableName)

- USEREF(DescriptionName;VariableName)

In the above statements 'VariableName' refers to the content of the `name` element of the variable we desire to reference, and 'DescriptionName' refers to the `descriptionName` element of the description that the variable we desire to reference is located in. It is assumed that the 'VariableName' is unique within the description it is located in and that the 'DescriptionName' is unique within the set of open descriptions.[17] (It is also assumed that the 'VariableName' and 'DescriptionName' do not contain semi-colon (';') character.)

A point that requires some clarification is how references involving variables that are located within a `block` construct should be handled (as these variables can have more than one value). In the current implementation references involving a block of multiplicity and that are not being made from a constraint are handled in the following manner:

- Reference from within a block to a variable outside a block - there is no issue, the reference resolves to the outside variable's unique value.

- Reference from outside a block to a variable inside a block - this does not resolve as the reference is ambiguous.

- Reference from within a block to a variable within the same block - the resolution of this depends on "context". If the referenced variable exists in the same structure as the referencing variable or exists uniquely in its ancestry then the reference is resolved accordingly. Otherwise, this does not resolve as the reference is ambiguous.

- Reference from a `numOccurs` element in a block to a variable within the same block - see above, there needs to be a clear context for this to resolve.

[17]If either is not unique the resulting behavior of an LCML processor will depend on its implementation. It could resolve to the first instance of the reference that it encounters or warn the user of the occurrence of an ambiguous reference.

- Reference from within a block to variable in another block - this does not resolve as the reference is ambiguous.

The manner of handling references from within constraints that involve multiplicity is slightly different:

- Reference from a single `item` element of a test is to a variable inside a block - this test has a true result if the tested relationship is true for all instances of the repeated variable.[18]
- References from both `item` elements of a test are to variables inside a block - This test can be evaluated if both referenced variables are in the same block of multiplicity and there is "context"[19] between the variables. The test has a true result if the relationship specified between the test items is true for every instance of shared context between the values.[20] Otherwise, this test does not resolve as the relationship between the referenced variables is ambiguous.

Functions

There are two functions supported by LCML. These functions can contain references and can be used in any element that a reference can. They provide a means for extending the usefulness of references.

- **CONCAT**(*item 1*;...;*item n*) - Used to join multiple *items* into an output string, where an *item* is either a string or a reference. *Items* are separated by the semi-colon (';') character. For example:
  - 'CONCAT(Hello; World;!)' yields the resultant string 'Hello World!'.

[18] For example, if we test whether a repeated string variable is not equal to 'bad value', the test returns true, if none of the values of the repeated string variable are equal to 'bad value'.

[19] If variables are in the same structure, they have a clear shared context and the relationship is expected to be true for each instance of that structure. If the variables are not located at the same level, but one is nested within the ancestry of the other there is also a context. In this case the relationship is tested between each instance of the variable that is deeper in the nesting and the variable that it is nested under.

[20] For example, if two repeated variables are located in the same structure, a test between them would be true, if the tested relationship was true for every repeat of the structure.

– 'CONCAT(Hello ;REF(myName);!)' yields the resultant string 'Hello Stephen!' (assuming the value of the variable "myName" is 'Stephen').

- **EVAL(***item 1*;*operator*;*item 2***)** - Used to evaluate a basic mathematical expression, where *item 1* and *item 2* can each be either a numeric value or a reference to a numeric value, and *operator* can be one of the characters '+','-','*','/', or '^' (where the meaning of these characters corresponds to their normal mathematical significance).

3.2.12 The Use of Formatting Elements

The `startText`, `endText`, and `separator` elements are provided for the "formatting" of output. The `startText` and `endText` elements are only included in output if a variable that is located in the content they are associated with[21] has been "used".[22] The use of the `separator` element is similar, in that it is only included in between instances of output where a variable has been "used". Similarly, the `header` and `trailer` elements are only included if the variable that they are directly associated with has been "used".

3.2.13 The Display of Hidden Variables

If all variables contained within a set or structure are hidden[23] then the set or structure should not be displayed to the user by an LCML processor. It becomes in effect hidden, because it has no variable content that can be made visible.

3.2.14 The Resolution of a Variable's Value

The output of a variable is resolved in the following order:

[21]The "used" variable need not be in the first layer of content, it can be nested some layers down.
[22](The output of the content can be blank, but at least one variable must be "used".)
[23](If a stucture or set contains a structure that has a non-hidden child variable, then not all variables are hidden.)

- If the value involves a reference that reference is resolved (and if the referenced value is itself a reference, it is first resolved and so on).[24]

- If the `uneditable` or `enumeration` elements have been used a variable's value should be validated against them.

- If the `aliases` element has been used, any relevant alias substitutions should be applied.

- The remaining (type-specific) validation should be done. If the type is numeric and the text starts with an equal sign ('='), this step should be skipped, and the equal sign removed.

- If the type of the variable is 'file' and the `fileType` element has a value of either 'includeFile' or 'template', then the variable's output should become the text of the indicated file.

- If referenced, a variable should return its output from this point.

- The `header` and `trailer` elements should be applied, if they exist. The variable output should now be: the text of the `header` element (if any), the output of the variable as resolved to this point, and the text of the `trailer` element (if any).

- The variable's output should be placed in the output text and then any applicable `replacement` elements under the `descriptionReplacements` element should be processed.

3.2.15 The LCML Schema

A schema has been developed for the Legacy Computing Markup Language using the XML Schema language [30] and is included as Appendix B. By using this schema in

[24] With this scheme it is possible to create a situation where there is a circular reference (where something makes a reference to itself). Such a reference has no significance and should not be expected to resolve successfully.

conjunction with a program that validates XML against a schema, an author of an LCML description can test a description to ensure that it meets the format defined in the schema. The LCML schema supports the inclusion of the fifteen Dublin Core elements by referencing a copy of the Dublin Core metadata schema.

Note that: the following three items are criteria for a proper LCML description file that the schema is unable to verify:

- A "type-dependent" element should only be included as a child of a `var` element if its use is compatible with the value of a variable's `type` element.
- The number of `dataInstance` elements in a `data` element should match the value of the `numOccurs` element for the corresponding structure.
- The number and order of `value` and `data` elements in a `dataInstance` element should correspond to the number and order of `var` and `structure` elements for the corresponding structure.

3.3 Discussion of the LCML and its Development

In this section we will discuss some of the issues that were considered in the development of the structure and syntax of LCML.

3.3.1 The Inclusion of Dublin Core Metadata

We have chosen to include the Dublin Core elements as optional elements in LCML descriptions for the purpose of documenting LCML files. There are several reasons to do so. It allows us to support existing metadata work and it provides a means to document LCML files using a mature standard. Presumably, this would allow for LCML descriptions to be incorporated into a digital document storage system at a future date. For example, a repository of all publicly available LCML files could be established and would potentially be indexable and searchable via the Dublin Core metadata.

3.3.2 Use of Elements vs. Use of Attributes

One of the questions that was considered during development of the LCML specification was: where in a description is it best to use XML elements and in where in a description might it be better to make use XML attributes? This is a common question in XML development and there are no clear, definite rules for making the distinction [26].

After some examination of the question, the decision was made to specify LCML descriptions entirely using elements and not to make use of attributes (with a few exceptions where uses of attributes are inherited from other specifications).[25]

The decision was based on the following observations and criteria:

- Attributes are best suited to the encoding of string values (and better suited to short strings at that).
- The use of attributes is not suited to describing data that could have a child element as content.
- In general, the use of elements produces XML that is more human-readable than similar XML that also makes use of attributes.[26]
- There is some benefit to minimizing the variety of forms of XML syntax that are used in encoding (and therefore decoding). This can simplify both description writing and processing.
- Consistency of syntax was an important consideration in design of the LCML specification.

Based on these criteria, there were only a few places in the specification where it seemed worth considering the use of attributes. Given the small number of cases, it

[25]Attributes are used by the root `description` element to reference the location of the LCML schema and the Dublin Core metadata elements all support the `xml:lang` attribute. These cases are uses of attributes that are inherited from other specifications, and are therefore in a sense only tangentially part of the LCML description.

[26]At least in the author's opinion.

was decided to develop the specification consistently using only elements (with the inherited exceptions that were mentioned previously).

3.3.3 Use of One vs. Many Types of Descriptions

Conceptually, the encapsulation of a program[27] can involve a number of different tasks. At the most basic level we want to be able to describe:

- How to run a compiled binary of our program (and what command-line options can be used).

We may also want to describe:

- How to generate a binary from its source code (invoking make, imake, cmake, configure, etc. as appropriate).
- How to write a makefile[28] to build a binary. (Makefiles are used to automate the process of building a program).
- The compile-time parameters associated with building a binary. These may be located in a file other than the binary's makefile such as a configurable include file. (Examples of parameters include an array size that is determined at the time of compilation and set via an include file or a "pre-processor define" flag).
- Details of the values of the parameters with which a binary was compiled.
- Input files associated with a binary and the runtime parameters within them.
- A structure that will organize the various descriptions associated with a program. (This could be one structure for all the descriptions relating to a program, or a sub-structure that categorizes all the descriptions of a certain category (e.g.

[27]While the encapsulation of a 'legacy' program is our primary goal with LCML, it is also not difficult to envision a scenario where we only want to describe the generation of a text file or script and don't really have the "encapsulation" of a specific program as our intent. For an example, see Section 3.5.3, where there is a discussion of using LCML to describe how to write an LCML description file.

[28]Or equivalent depending on the system employed.

we might categorize the 'source' descriptions which are used to build a binary, and refer to them all from one 'source' description).

At an earlier stage of this work we used an approach that contained a number of different types of description files to handle these different tasks [7]. We had the following types of descriptions: *Program*, *Source*, *Makefile*, *Binary*, and *Parameter*. This is clearly not the only possible set of description types that could be created (as evidenced by the successful switch to a single description type), but it proved to be a workable set.

The decision to create a single type was made upon observing significant overlap in the functionality of different description types:

- The *Program* and *Source* types were variations on a similar theme and a variation of their core functionality was already present in the *Binary*, *Makefile*, and *Parameter* types.

- There was significant overlap in functionality between both the *Makefile* type and the *Binary* type, and the *Makefile* type and the *Parameter* type (the entire functionality of the *Parameter* type was a actually a subset of the functionality of the *Makefile* type).

And in addition it was noted:

- In basic terms, descriptions have two tasks: the creation of text output from parameters, and the structuring of relationships between descriptions and parameters.

- There can be significant advantages to re-use of programming effort (this is true for both XML and Java development).

- There are benefits to having to deal with only one description type, rather than five distinct description types, even if the one description type is slightly more complicated as a result.

- While restructuring the description types into a single description type was expected to be a significant undertaking, a substantial increase in flexibility was expected to result.

- The only apparent drawbacks to a restructuring were: the effort involved, and the possibility of increased complexity in the description of some aspects of the encapsulation.[29]

It was therefore decided to restructure the encapsulation format making use of a single type of description file. The *Parameter* type was used as the basis for this new description type. While some parts of the restructuring involved a natural transition, other portions required more careful consideration.

The result is that we have a single *description* type that is designed to describe how to assemble sets of adjustable parameters into information rich output text. The resultant output can be saved as a file or treated as a script. The same description type can be used to describe relationships between parameters and descriptions. This set of functionality provides the capabilities necessary to encapsulate command-line driven programs.

3.3.4 The Design of a Variable-Based Description

Another design decision that was made was to make descriptions variable-centric.[30]

There were two implications of this decision:

- an LCML processor is only expected to implement an *editable* user interface for the values and settings of variables.

- since nearly all the encapsulation of parameters (and all the encapsulation of *editable* parameters) is done through the use of variables, their implementation needs to be done in a manner that is flexible enough to describe the information that they need to encapsulate.

[29] Upon completion of the restructuring it appears that while some complexity has been added, most of the additional complexity corresponds to increased functionality.

[30] An alternate approach was considered making use of both "variables" and "objects", but was abandoned after seeing no real benefit to such a distinction.

The resulting framework uses variables coupled with references to allow for a lot of flexibility. A key factor of this framework, is that not all variables have to directly contribute to a description's output (as their "use" setting can be 'false'). This means that variables can be used both to describe output or to describe data relevant within a description.[31]

3.3.5 The Separation of Encapsulation and Processing

A design goal of the LCML specification, was that its implementation should stand independently from the implementation of a processing program. Remember that in order to make use of an LCML description file one needs to process it somehow.[32] In other words, we have a two-tier architecture, with *encapsulation* and *processing* being separate tiers.

There are several reasons that encourage such a separation between encapsulation and processing. Such an architecture:

- is a natural extension of the use of different technologies. (XML for encapsulation and Java for processing).
- facilitates the individual development of both tiers.
- effectively permits the creation of multiple processing programs, or multiple versions of a processing program (for example, a JavaScript version of a processing program could be used alongside a Java version with the same set of description files).

A practical consequence of this approach is that the LCML should not describe specifics of the user interface it expects to rendered to. For example, while we can encode a list of choices that a variable's value could have (using the `enumeration` element), we do not specify from LCML that is should be displayed in a combo-box, or that its display should have certain dimensions.

[31] For an example of flexibility with the current arrangement, a description author can stage a reference through several variables applying aliases, functions, and constraints along the way.

[32] The LEGEND program, discussed in Chapter 4, is an example of a tool that is under development to process LCML descriptions.

There is an element in the LCML description specification where this separation is not strictly enforced: In our current implementation it is left up to a processing program to interpret the text of the `command` element.[33] There isn't really an effective, implementation-independent way to describe the launching of scripts, as different platforms, command-line shells, or processing programs may use a different syntax. Rather than restrict a user to a single "standardized" method and shell to intially launch a script, it was decided that the description author should be able to specify the command used to run a script. The flexibility gained was considered more significant than breaking the separation between encapsulation and processing otherwise implemented.

3.3.6 Multiplicity

The term 'multiplicity' is used as a general term to describe cases where a parameter or collection of parameters has a multiplicity of values. The LCML language handles multiplicity with the `block` construct.

Multiplicity can take a wide variety of forms; describing them and handling their implications proved to be one of the most challenging aspects of developing the description format. The simplest example of an instance of multiplicity is that of a one-dimensional array of values for a single variable. A slightly more complicated example of an instance of multiplicity is a two-dimensional array, and the most complex example is that of a jagged, multi-typed, n-dimensional data structure where individual variables can be disregarded at will. We will consider each of these requirements in turn, and in some detail.[34]

[33] Currently LEGEND interprets it using the Runtime.exec() Java method. Similar methods should be available in other programming languages (and may even work with the same or very similar syntax to what is used with the Runtime.exec() method.)

[34] This last example can also be thought of as a tree-like data structure of non-homogenous nodes, where nodes are expected to have certain characteristics based on their location in the tree. The structure we have developed is not suited to highly arbitrary data that differs greatly in content and contains little structure. It is probably fair to note though, that generally the parametization of scripts or input files to programs does not exhibit extreme instances of these characteristics. There probably are also imaginable cases that though supported by our structure, start to stretch its usefulness.

Describing n-dimensional data

The LCML implementation supports an n-dimensional data structure by the recursive nature of the `structure` and `data` elements. This allows for an unlimited amount of nesting. While in practice it is expected that *most* descriptions will be limited to two or three levels of nesting, given the recursive nature of specification supported by XML, it is not really any more complicated to support an n-dimensional nesting than an explicit two or three dimensional nesting.[35]

Jagged

We are using the term "jagged array" to describe a multi-dimensional array that is not necessarily rectangular. A two-dimensional "jagged array" can be thought of as a vector, each element of which is a vector containing an arbitrary number of objects.

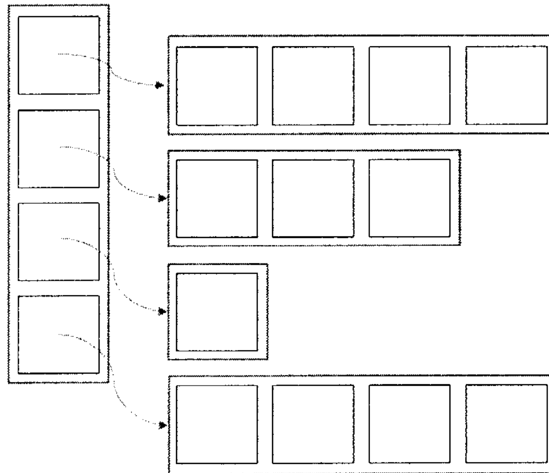


Figure 3-2: Depiction of a Two-Dimensional Jagged Array

The primary difficulty that supporting the description of jagged arrays adds is the question of how to specify the individual indices. The following example demonstrates the recommended method, which makes use of a reference:

[35]It is worth noting that there are also challenges in viewing and editing n-dimensional data, and that an LCML processor needs to implement some form of "recursive" display, to appropriately handle the recursive data structure.

```
<block>
  <structure>
    <structureName>Example Structure</structureName>
    <structureInfo>Demos a 2D jagged array.</structureInfo>
    <numOccurs>2</numOccurs>
    <var>
      <name>jaggedIndex</name>
      ...
    </var>
    <structure>
      <structureName>Jagged Structure</structureName>
      <structureInfo>Dimension = 'jaggedIndex'</structureInfo>
      <numOccurs>REF(jaggedIndex)</numOccurs>
      <var>
        <name>My Letter</name>
        ...
      </var>
    </structure>
  </structure>
  <data>
    <dataInstance>
      <value>1</value>
      <data>
        <dataInstance>
          <value>A</value>
        </dataInstance>
      </data>
    </dataInstance>
    <dataInstance>
      <value>3</value>
```

```
      <data>
        <dataInstance>
          <value>A</value>
        </dataInstance>
        <dataInstance>
          <value>B</value>
        </dataInstance>
        <dataInstance>
          <value>C</value>
        </dataInstance>
      </data>
    </dataInstance>
  </data>
</block>
```

Multi-typed

The idea of multi-typed array[36] is handled in a straight-forward manner by the `structure` construct. A multi-typed array can be described in a straight-forward manner by placing variables with the desired types within a single structure. This can also be combined with a nested structure to support a more complex situation such as the inclusion of a vector within a multi-typed array.

Disregarding Variables

In the design of the structure of multiplicity we have included support for a multiplicity of values, but not a multiplicity of "use" settings. There are however, cases where it might be desirable to be able to "use" a variable on some repeats and not "use" it on others. There is a way to handle this through references. The "use" setting for the variable whose value we desire to control should contain a reference to a variable

[36]Extensive work was done investigating how to represent one-dimensional and two-dimensional arrays within a variable before it was concluded that a multi-type array was truly necessary to represent the variety of situations that we expect could be encountered.

that is located in the same structure and whose value evaluates to 'true' or 'false'. The referenced variable can then be edited, thereby controlling the "use" setting of the referencing variable (the "referenced" variable can have its own "use" setting set to 'false' thereby having no impact on the output of the description).[37] The following is an example:

```
<block>
  <structure>
    <structureName>Example Structure</structureName>
    <structureInfo>Demo of disregarding variables.</structureInfo>
    <numOccurs>1</numOccurs>
    <var>
      <name>myVariable</name>
      ...
      <use>Ref(useMyVariable)</use>
      ...
    </var>
    <var>
      <name>useMyVariable</name>
      ...
      <enumeration>true;false</enumeration>
    </var>
  </structure>
  <data>
    <dataInstance>
      <value>myValue</value>
      <value>false</value>
```

[37]This arrangement can also be coupled with the use of aliases and multiple references to allow a non-boolean setting to be "filtered" to a boolean setting to control the use of a variable, or to control the "use" of a number of variables with one edit. There are scenarios where either of these techniques could prove useful. (This is a tangential statement, as this could be useful outside of a block of multiplicity as well).

```
    </dataInstance>
  </data>
</block>
```

Multiplicity of Binaries and Files

There is no problem directly associated with including a variable of the 'binary' or 'file' type within a multiplicity stucture. However, we have not supported a strong linking between child descriptions and variables in multiplicity. (Or any variables for that matter as the linking we have supported is informational through the `association` element or implied through the use of references between descriptions). In other words, the concept of multiplicity has not been extended to cover a multiplicity of descriptions.[38]

The Separation of Structure and Data

Our approach to describing multiplicity splits the description into `structure` and `data` elements. We considered the alternative of not separating structure and data and mixing them into a single representation. After some examination it was concluded that separating the structural and data representations resulted in a cleaner and overall more understandable representation.

3.3.7 Encapsulating Information

A portion of the process of encapsulation deals with capturing the knowledge that is necessary to use a program. While some of this knowledge becomes embedded into the validation of variable values and the description of constraints, there is also the possibility to provide both general and specific information to the user. In the

[38]While we have not made any special provisions to support multiple of instances of the same description file, we can approach a situation where multiple instances of the same description are required by independently declaring each instance of a description file (over-declaring if necessary) or by using a single description file to sequentially write multiple files. While neither of those suggestions may seem ideal, the complications involved in the implementation of (and particularly the meaningful referencing of) a multiplicity of descriptions are significant and such a capability is beyond the scope of the current work.

LCML specification there is an element for information provided at nearly every level of the description: `description`, `set`, `structure`, and `variable`. As the text of these elements is intended to be provided to users of encapsulated programs, making effective use of these elements can go a long way towards helping a user learn and use an encapsulated program.

It is also worth mentioning that since the informational elements simply contain text, there is a bit of freedom in exactly what information is described and in what manner it is described. With some thought and effort, within a well-developed description it should generally be possible to encode much of the information present in a program's documentation into the information elements that provide help to the user. If this is done, a user can work with the encapsulated program with less of a need to reference its documentation.

It is recognized that there are instances where an information element is required by LCML but there may not really be relevant information to place there. Despite the fact that this can occur, the information elements were made mandatory as it is presumed that if they are required there is a greater chance they will be used effectively; and even if only a little information is provided, in many cases a little help is preferable to no help.

3.3.8 The Use of Validation

One of the difficulties found with the traditional command-line driven interface is that the only opportunity for the "validation" of input parameters is typically whatever validation is done by the program at runtime, i.e. once it is already running. Furthermore, the process of editing the scripts and arguments used by command-line driven interfaces can be error-prone. This is true in terms of logical errors (specification of an incorrect or inappropriate value), syntax errors (forgetting an extra space, comma, etc.), and typographical errors (editing of text with accidental unintended results). Because of the problems associated with this scenario a more capable system of data validation should prove helpful to a user. This is especially true in the case of a program that may take hours to run (and crash at some point in its run), or

in a scenario where programs are being launched into a queueing system (where the results will be collected later).

With an encapsulation approach such as LCML there is the opportunity to make provision for a more agressive validation scheme that ensures the validity of the data against basic criteria. These criteria can be applied while a user is editing the parameters of a description. There are also many cases where it is desirable to "validate" parameters in terms of their relationships to other parameters in the input file or against more extensive criteria. While this is similar in concept to basic validation we have separated it conceptually, and handle it under the term "constraints".[39]

It is worth noting that the XML Schema language provides a fairly extensive set of functionality for validation. Some of this functionality overlaps concepts we have implemented (for example the specification of a numeric range). We have not made use of this functionality, and there are several reasons not to:

1. The XML Schema based validation is not expressive enough to handle all the situations we would like to express.

2. Making use of the XML Schema language would require restructuring our descriptions. A description author would then have to learn both the constructs of the XML Schema language, and our adaptation of it for use in legacy encapsulation (if such an adaptation is even possible).

3. Performing our own validation affords us the opportunity (and on the downside requires us to) provide our own handling of validation errors.[40]

In LCML, the following properties can be specified at the variable level and can

[39]We actually have three concepts that are associated with the idea of "validation".

1. The validation of an LCML description against a schema (i.e. schema-validation).
2. Basic validation of variable values as supported by LCML (what we are currently describing).
3. Validation of specific criteria and relationships between variables (what we have labeled "constraints").

These concepts are distinct and should not be confused.

[40]If we had used an XML Schema-based approach we could make use of standard routines for schema validation.

be tested for validity (the LCML elements used to specify the property are included in parenthesis):

- A variable's value cannot be edited. (`uneditable`)
- A variable's "use" setting cannot be edited. (`uneditable`)
- A variable's value must be part of a list of acceptable values. (`enumeration`)
- A 'numeric' variable's value must be a number.[41] (`type`)
- A 'numeric' variable's value must match an expected precision (integer, float, long, double).[42] (`type`, `precision`)
- A 'numeric' variable's value must fall within a certain range.[43] (`type`, `range`)
- A 'string' variable's value must contain a minimum of a certain number of characters. (`type`, `minLength`)
- A 'string' or 'file' variable's value may contain a maximum of a certain number of characters. (`type`, `maxLength`)

3.3.9 The Use of Constraints

The basics of the use of constraints have been described in some detail in Section 3.2.9. The system was developed to be able to reflect complicated relationships between more than one variable. The encapsulation of constraints is verbose, but here it was decided that clarity is preferrable to brevity.

One difficult situation that is related to the use of constraints, is how to handle an instance where a constraint involves the options that a binary was built with.[44] The following scheme is proposed as a method for handling such a situation:

[41] If a numeric variable's value begins with an "=" then validation is bypassed (this allows an expression to be used).
[42] See previous note.
[43] See previous note.
[44] This scenario is certainly applicable to our test case HOPS.

- A description can be written to accompany a binary description. This description should output a file in *LCML* description format[45] which describes any parameters that may be referenced after the binary is built.[46]

- This description can make use of references to retrieve the values of relevant parameters from the binary description.

- At build-time, a copy of this "build-time" description can be saved.

- Constraints can be then written involving these stored "build-time" parameters.

It is also worth noting that describing *all* constraints imaginable is not necessarily a worthwhile or even possible task. Constraints can certainly assist a user in running a program, but at some level the user has to understand how to use the program that has been encapsulated. There are probably limited returns for the effort involved in describing esoteric constraints. Furthermore, it is worth remembering that guidance can be provided to the user through the information elements, and that this can be used to guide the editing process. While this clearly does not provide any form of checking, it does provide a means of helping the user to use the encapsulated program properly (and it can be used to inform the user of constraints that could not otherwise be described with LCML).

3.3.10 Restricted Descriptions

One of the ideas incorporated into LCML is an allowance for the restriction of the visibility of variables in a description.[47] This is included because there are cases that it might be desirable to restrict a description such that a user does not need to view all variables. (See [17], where usability testing of a command-line encapsulation system resulted in a similar conclusion).

[45]Here we are proposing the use of LCML to describe an LCML file because we can easily parse and read the parameters stored in an LCML file at a later time.

[46]The `absoluteLocation` element was included in LCML so that we would be able to read back in such a description.

[47]While this restriction of visibility (i.e. 'hiding" variables) is described in LCML, it would have its effect when the description is being viewed and edited with an LCML processor.

A restriction can be done purely for the sake of convenience (it may simply be unecessary to view some variables) or it could be done as a means of limiting a user to a subset of the functionality in a description. (For example, this could be used to prevent a novice user from incorrectly editing parameters that do not require any attention or to provide a novice user with a simpler interface). With the scheme described here, a description writer can simply provide different states of descriptions (you could have a 'spring', 'summer', and 'fall' version of a description) or could specifically envision user roles such as 'novice', 'intermediate', and 'expert'.

In LCML, the restriction of a variable's visibility is accomplished using the `hidden` element. As discussed in Section 3.2.6, a `hidden` element contain either the text 'true' or 'false' or a reference that resolves to 'true' or 'false'. If the hidden setting is 'true' then the element should not be made visible to the user by an LCML processor.[48]

Because the `hidden` element can contain a reference, it is possible to setup a dynamic description where variables can be hidden and unhidden. Such a scenario could be used to hide parameters whose relevance depends on other parameters or to allow the display of multiple description states (as discussed above) from a single description.[49]

3.3.11 The Use of Replacements

There are two functionalities that can be used for "replacement" (i.e. the substition of text) in LCML. One occurs on the description level and makes use of the `descriptionReplacements` element. The other occurs on the variable level and makes use of the `aliases` element. While they provide some overlap in functionality, they also can be useful in different cases.

Replacements can be used in a variety of ways, some of which have been suggested

[48]Though an LCML processor could also support a "show hidden variables" mode, where all variables are shown regardless of their "hidden" setting. Such a mode could also give a user the capability to edit the "hidden" settings of variables. (There are some benefits to allowing such a mode and we have implemented one in LEGEND).

[49]If the creative use of `alias` elements is incorporated into such a scenario (being used inside "helper variables" to filter a "mode" string into a 'true'/'false' value as appropriate), a description author could create a description where a user can choose between one of several display "modes" by editing a single variable.

already. We will illustrate one additional manner here.

A question that arose was: when we encapsulate a program that makes use of a number of files, is there any way to describe the file paths in a manner that they can be easily moved from one computing account or environment to another without having to individually reconfigure each path. (While the use of relative paths could provide a partial answer to this, it isn't a perfect solution). The following is an approach using aliases.

- Within each file variable of interest a description author can specify the path in terms of some common base value, '$BASEPATH/mysubdirectory/myfilename' (where 'mysubdirectory' and 'myfilename' correspond to the directory and file of interest).

- The description author can also specify an "alias", where the "alias key" should be '$BASEPATH' and the "alias output" should be a reference to a file variable which contains the "base path" as its value.

- All the file variables can be made to reference a common variable containing the base path in this manner.

- In this scenario, if the code is installed on a new system, the description can be updated by changing one variable. The use of the alias replacement type automatically extends the change to wherever it needs to be implemented.

  Note that: here we are assuming that the same file structure is used on the new installation (just with a different base path).

3.3.12 Creating and Running Scripts

One of the goals of LCML was to be able to not only encapsulate the input files used by a legacy program but also to be able to encapsulate the commands used to run a program and the process used to compile a legacy binary. The language was developed to be suited to both of these possibilities.

One of the questions that came up was how to best generate scripts, and a limited deterministic approach was rejected in favor of a more flexible approach (scripts and files are now described in the same manner, so all the flexibility that was developed to describe parameter files can now be applied to scripts). The approach used should be able to work with a very wide variety of command-line driven systems. For example, the current framework should be flexible enough to describe: the UNIX 'cp' command, launching an program using MPI (Message Passing Interface), using Sun Grid Engine to queue a script for running a legacy program, or the compilation of a program with a tool such as 'gmake'.

3.3.13 Storing State - Import/Export Format

The values and settings of variables in an LCML description provide a default state for the description; however, it is also desirable to be able to save other states of the description (with different values and settings). One manner in which this can be done is to simply replicate the entire description, but with an edited set of values and settings. While this can effectively capture the state of a description, and is a straight-forward manner by which to create multiple versions of a description, three drawbacks have been identified to this approach:

1. It is a verbose approach. Much of the information contained in a description is static, and immutable by the process of editing the description. It is unecessary to duplicate this information every time the state of the description is stored.

2. It works on a description level. When editing a set of descriptions (as would be typical for the editing of an encapsulated program), it may be desirable to save the state of all descriptions, or a group of descriptions, and not have to consider them individually.

3. It is a static approach. It is reasonable to expect that as encapsulated programs undergo continued development that the descriptions that describe them may require incremental updates. A previously saved copy of a description would

become incompatible and need to edited "by hand" for use with the updated program. However, if a file only stores the state of the program, it could be used to modify the state of an updated description to match the stored state (as best as possible).

So, an alternative approach is to develop a format for storing the state of the variables in a description. It is not intended that a file in such a format would be written by a person, but rather that it would be generated by an LCML processor such as LEGEND, which could have the capability to export and import description states.

3.4 Example Descriptions

3.4.1 Hello World Example

We'll start with a traditional "Hello World" example. The following LCML description can be used to create a text file that contains the text "Hello World".

```
<?xml version='1.0' encoding='UTF-8'?>
<description
    xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
    xsi:noNamespaceSchemaLocation='http://deslab.mit.edu/LCML/lcml.xsd'>

  <descriptionName>Hello World Example</descriptionName>
  <descriptionInfo>This example is used to generate a file
                  that contains the text ''Hello World''</descriptionInfo>
  <descriptionTarget>
    <file>HelloWorld.txt</file>
  </descriptionTarget>
  <descriptionContent>
    <set>
      <setName>Hello World Set</setName>
```

```
      <setInfo>This set is used to hold the Hello World variable.</setInfo>
      <var>
        <name>helloWorld</name>
        <info>This variable is used to output the text "Hello World".</info>
        <type>string</type>
        <value>Hello World</value>
        <use>true</use>
        <hidden>false</hidden>
      </var>
    </set>
  </descriptionContent>
</description>
```

Notes:

- The first line is typical and indicates the XML version and character encoding used by this LCML document.
- In this example, the root `description` element contains two attributes. These are used to reference the location[50] of the LCML schema. (A local path to a copy of the schema could also be substituted).
- We have used the `descriptionTarget` element to declare that this description can be used to create file output and that the created file should be named named "HelloWorld.txt".
- We have used a variable within the `descriptionContent` section to encode the actual text of the file, "Hello World".

3.4.2 Advanced Hello World Example

The following is a more complex "Hello World" example.

[50]The location listed here is not guaranteed to be the final online location for this schema.

```
<?xml version='1.0' encoding='UTF-8'?>
<description
    xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
    xsi:noNamespaceSchemaLocation='http://deslab.mit.edu/LCML/lcml.xsd'
    xmlns:dc="http://purl.org/dc/elements/1.1/">

  <descriptionName>Advanced Hello World Example</descriptionName>
  <descriptionInfo>This example is used to generate a file that says
                 Hello to the world and to the user, and
                 demonstrates some features of LCML.</descriptionInfo>
  <descriptionMetadata>
    <dc:title xml:lang="en">Advanced Hello World Example</dc:title>
    <dc:creator>Stephen Geiger</dc:creator>
  </descriptionMetadata>
  <descriptionTarget>
    <file>HelloWorld.txt</file>
  </descriptionTarget>
  <descriptionContent>
    <set>
      <setName>Hello World</setName>
      <setInfo>Holds the basic Hello World.</setInfo>
      <var>
        <name>helloWorld</name>
        <info>This variable is used to output the text
              "Hello World
               and Hello USER!" </info>
        <type>string</type>
        <value>Hello World</value>
        <use>true</use>
        <hidden>false</hidden>
```

```
      <trailer>&#x000A;and Hello USER!</trailer>
      <uneditable>value;use</uneditable>
    </var>
  </set>
</descriptionContent>
<descriptionReplacements>
  <replacement>
    <replacementKey>USER</replacementKey>
    <replacementContent>
      <set>
        <setName>User Name</setName>
        <setInfo>Contains the user name.</setInfo>
        <var>
          <name>userName</name>
          <info>Enter your name here.</info>
          <type>string</type>
          <value></value>
          <use>true</use>
          <hidden>false</hidden>
        </var>
      </set>
    </replacementContent>
  </replacement>
</descriptionReplacements>
<descriptionConstraints>
  <conflict>
    <test>
      <item>REF(userName)<item>
      <relation>SAME</relation>
      <item>Sam</item>
```

```
    </test>
  </conflict>
</descriptionConstraints>
</description>
```

Notes:

- We have added an additional attribute ('xmlns:dc') to the root `description` element. It is used to define the 'dc' namespace and allows us to reference the Dublin Core metadata elements.
- We have added an example of the inclusion of two of the Dublin Core metadata elements as a means of documenting our description. The first of the two Dublin Core elements we have included demonstrates the use of 'xml:lang' attribute.
- We have added the use of a trailer element that adds the text 'and Hello USER!' to the variable output. The text '`
`' is a representation of a newline character so that 'and Hello USER!' is put on a new line.
- In this example we have used the `uneditable` element to indicate that the user should not be able to edit the "value" or "use" setting of the helloWorld variable.
- In this example we have also used the `descriptionReplacements` element. In this case it is used to replace the text "USER" with an appropriate user name.
- We have specified a constraint that indicates that the supplied user name cannot be 'Sam'. The constraint also demonstrates the use of a reference to the "userName" variable.

3.4.3 'cp' Example

A more practical example is a description of the UNIX 'cp' command. In this case, instead of creating a file we want to be able to run a script.[51]

[51] For this example we have assumed that the script will be invoked by the 'runtime.exec()' method in Java. This assumption affects the syntax of the `command` element, though the syntax we have

```
<?xml version=’1.0’ encoding=’UTF-8’?>
<description
    xmlns:xsi=‘http://www.w3.org/2001/XMLSchema-instance’
    xsi:noNamespaceSchemaLocation=‘http://deslab.mit.edu/LCML/lcml.xsd’>

  <descriptionName>‘cp’ Example</descriptionName>
  <descriptionInfo>This example is used to generate a script that copies
                a file using the UNIX ‘cp’ command.</descriptionInfo>
  <descriptionTarget>
    <script>
      <command>sh tempscript</command>
      <scriptname>tempscript</scriptname>
    </script>
  </descriptionTarget>
  <descriptionContent>
    <set>
      <setName>‘cp’</setName>
      <separator> </separator>
      <setInfo>This set is used to describe the ‘cp’ command,
               which is used to copy files.</setInfo>
      <var>
        <name>‘cp’ binary</name>
        <info>Used to write the ‘cp’ command in a script.</info>
        <type>binary</type>
        <value>cp</value>
        <use>true</use>
        <hidden>true</hidden>
      </var>
      <var>
```

used here could potentially work with other implementations as well.

```
      <name>Source File</name>
      <info>Enter the source file here.</info>
      <type>file</type>
      <value>mysourcefile</value>
      <use>true</use>
      <hidden>false</hidden>
    </var>
    <var>
      <name>Destination File</name>
      <info>Enter the destinations file here.</info>
      <type>file</type>
      <value>mydestinationfile</value>
      <use>true</use>
      <hidden>false</hidden>
    </var>
  </set>
 </descriptionContent>
</description>
```

Notes:

- We have made use of the `separator` element to insert a separator (in our case a space) between the output of the variables.

- We have hidden the 'cp' binary variable as there is no real reason to display it to the user.

- The encapsulation done here could be incorporated into a larger script, and used as a means of staging files while working with another program. This file staging could be before and/or after the other program is run.

3.4.4 SGE Example

A final example is how to write a description that can be used to launch a script in the Sun Grid Engine environment.

```
<?xml version='1.0' encoding='UTF-8'?>
<description
    xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
    xsi:noNamespaceSchemaLocation='http://deslab.mit.edu/LCML/lcml.xsd'>

  <descriptionName>SGE Example</descriptionName>
  <descriptionInfo>This example demonstrates how to
                   use LCML to run a script with the
                   Sun Grid Engine program. The script itself simply
                   contains a UNIX command.</descriptionInfo>
  <descriptionTarget>
    <script>
      <command>qsub tempscript</command>
      <scriptname>tempscript</scriptname>
    </script>
  </descriptionTarget>
  <descriptionContent>
    <startText>#$ -S /bin/sh&#x000A;</startText>
    <set>
      <setName>Script Content</setName>
      <setInfo>Holds a unix command.</setInfo>
      <var>
        <name>myCommand</name>
        <info>Used to launch a UNIX command.</info>
        <type>binary</type>
        <value>uname -s</value>
```

```
        <use>true</use>
        <hidden>false</hidden>
      </var>
    </set>
  </descriptionContent>
</description>
```

Notes:

- To launch a script into SGE we have simply used an appropriate syntax for the `command` element and content for the `startText` element that points the script to an appropriate shell to use from SGE. For more details on the general use of SGE, see the SGE documentation [11].

3.5 Tools for Generating Descriptions

The LCML document format is an XML-based document specification with some inherent complexity and syntax. In order to simplify the process of preparing LCML description files the possibility of providing an LCML description authoring tool was investigated. Such a tool could be be provided as an alternative to preparing all LCML descriptions directly in a text editor ("by hand"). The following are potential benefits for the description author:

- Reduced effort of physically entering the text of an LCML description.
- Readily available guidance as to the appropriate structure of the LCML description.

Three general approaches were considered:

1. Use of an existing XML editor with specific application to LCML description files through either validation against the LCML schema or custom modification/extension of the editor.

2. Extension of the features of an existing text editor by adding tools to simplify the process of authoring an LCML description.

3. Creation of an LCML description that is capable of assisting in the creation of other LCML descriptions.

3.5.1 Use of an Existing XML Editor

A search was undertaken for freely available (preferably open-source) cross-platform XML editors, that included support for XML-based schemas.[52] There was not much found that seemed to meet the criteria, and the best prospect among the XML editors investigated was Pollo [23], which does provide support for schema validation.[53]

The alternative idea of customizing an existing XML editor to LCML would likely require a significant programming effort and therefore this approach is not recommended given the possibilities present with the other two approaches.

3.5.2 Extension of a Text Editor

The second idea was to enhance an existing text editor to make it easier to write the text of LCML description file. With this approach, the description author would have to work with the "source view"(i.e. editing the raw text and tags) of the XML; however, the extensions could provide features that facilitate the insertion of skeleton LCML text, thereby simplifying and providing guidance to the editing process.[54]

There are a large number of text editors available and several candidates for extension are discussed here.

[52]Support for XML based schemas was considered an important characteristic because a well implemented XML editor could then provide editing assistance that is specifically relevant to LCML by referencing the schema. The editor could also then be used to ensure that the description produced was a valid description according to the schema.

[53]Schema validation in Pollo warns about the presence of an unexpected "xsi:noNamespaceSchemaLocation" attribute in the LCML `description` element. If this is encountered, it can be ignored.

[54]It is worth noting, that while working in such a view maybe somewhat intimidating at first to someone not real familiar with XML, with a little experience this could be a fast approach to writing descriptions.

EMACS

The EMACS editor, better known in the Unix world, is a text editor that is available on a wide variety of platforms. It supports "modes" which are used for editing specific types of text, and a possible approach would be the creation of a "mode" for the editing of LCML.

JEdit

JEdit [16] is a mature Java-based, open source, programmer's editor that has support for macros as well as a plugin architecture. It might be possible to write an effective plugin for the authoring of LCML descriptions.

Microsoft Word

Microsoft Word contains advanced and powerful macro/scripting capabilities. The creation of a set of tools for authoring LCML in Word would be relatively straightforward given a knowledge of the Visual Basic for Applications (VBA) scripting language, and an example of such an implementation was prototyped.

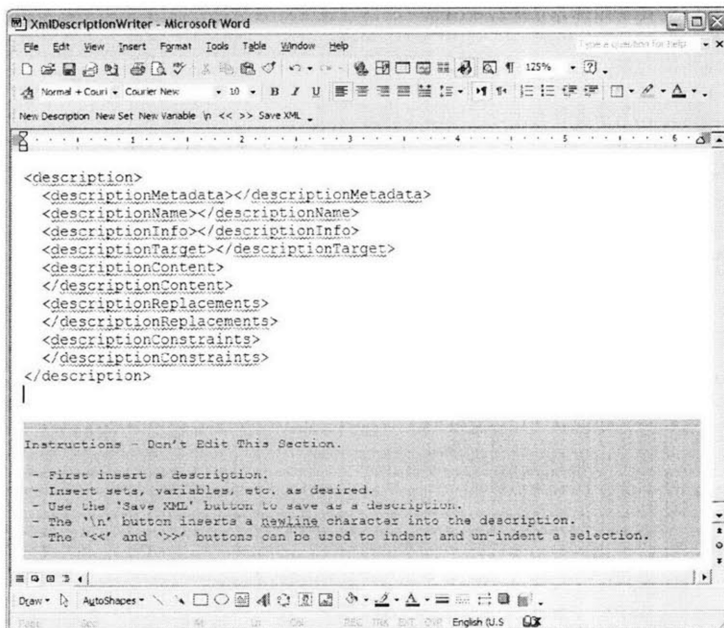


Figure 3-3: Prototype of LCML Authoring Extensions for Microsoft Word

However, such an implementation does not meet two goals present in the project: the use of tools that are available cross-platform and are freely available.

OpenOffice.org

OpenOffice.org [20] is an open source office suite that is available for a number of platforms and which contains macro and scripting capabilities. As such, its use should be suitable for the project. It is expected that an implementation of a set of tools to assist in authoring LCML could be done for OpenOffice.org's word processing package. The implementation could be made similar to the tools prototyped in Microsoft Word.

3.5.3 Use of LCML to Author Itself

While LCML has been developed for the creation of text relevant to legacy computing (primarily input files and shell scripts), it proves to be well-suited to the creation of information-rich text and it is therefore possible to create an LCML document that is designed to author other LCML documents.[55] A prototype of this technique shows promise.

[55] This also proved to be a good test case for LCML as in it we are forced to couple a high level of multiplicity with a level of variability of content within that multiplicity.

Chapter 4

LEGEND (LEGacy Encapsulation for Network Distribution)

In this chapter we discuss the development and use of the program LEGEND (LEGacy Encapsulation for Network Distribution). We will first overview the use of the program and then discuss the handling of several specific issues.

4.1 The Role of LEGEND

The LEGEND program is being developed as a tool for processing and displaying description files written in Legacy Computing Markup Language (LCML) and for generating output from them. LEGEND automatically generates a user interface for a set of LCML descriptions. A user can then edit parameter values (with assistance provided in understanding the parameters and in validation of the parameter values) or as appropriate create a file and/or a script output for a description state. If a script is created, it can then be run from the LEGEND interface. In this manner LEGEND can be used to run a program encapsulated in LCML.

4.2 LEGEND Program Overview

4.2.1 Getting Started

Since LEGEND works with Legacy Computing Markup Language (LCML) descriptions, the process of using LEGEND begins by selecting **Open ...** or **Open from web ...** from the LEGEND's **File** menu. This allows the user to open an LCML description. Alternatively, the user can open a file that was created by exporting a description state and the description that it references will automatically be opened (and its values and settings set to the exported state).[1] If the **Open from web ...** functionality was selected, a mini web-browser is presented, and the user can either "browse" to the location of a file or can directly enter the URL[2] of a file.

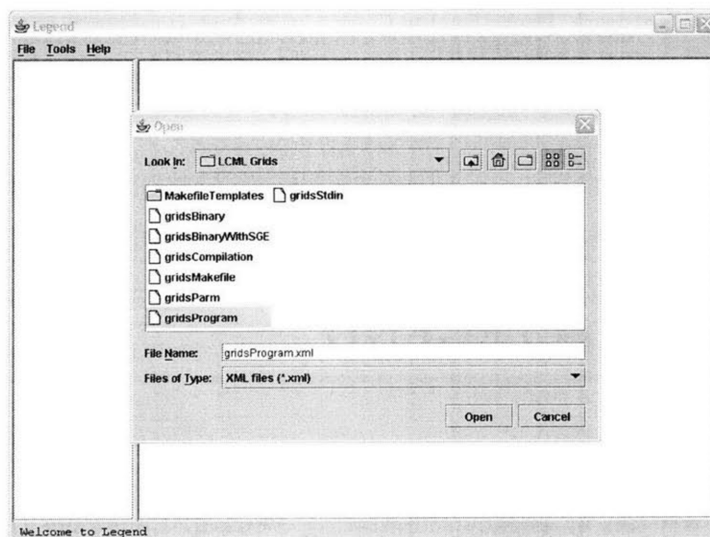


Figure 4-1: Opening a description file.

[1] This feature is undergoing development.
[2] Uniform Resource Locator.

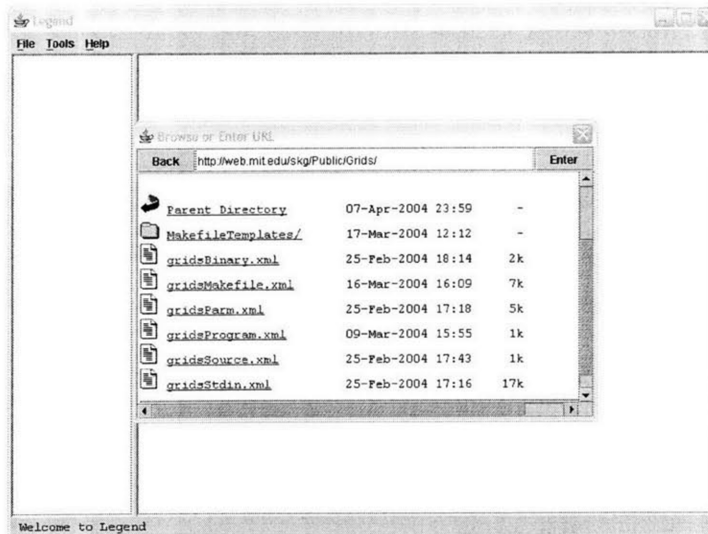


Figure 4-2: Opening a remotely located description file.

When a description is opened, it is opened along with any LCML "child" descriptions referenced by it, as well as any of their "descendant" descriptions.[3] The following is an example result:

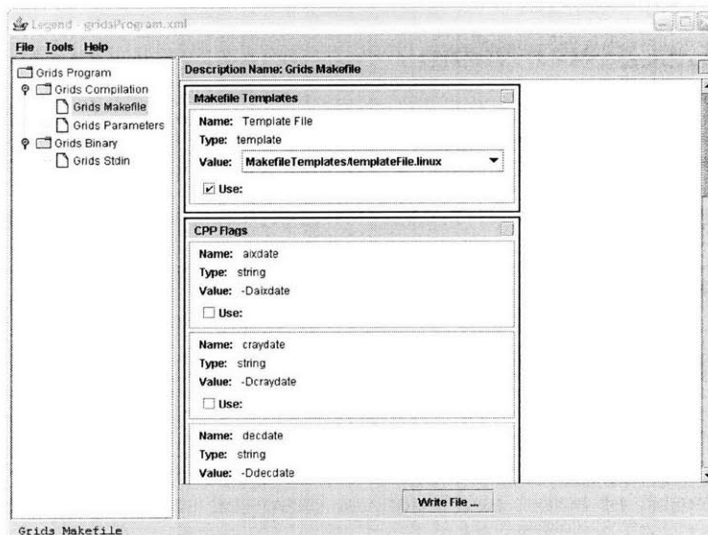


Figure 4-3: Example of a set of opened LCML description files.

[3]In other words, a description opens with all its child descriptions, which in turn open with all their child descriptions, and so on recursively (until no more descriptions are referenced).

There are four areas of the screen to consider: the menu, the LCML description tree, the LCML description display area, and the status bar. We will discuss the use of each of these areas in turn.

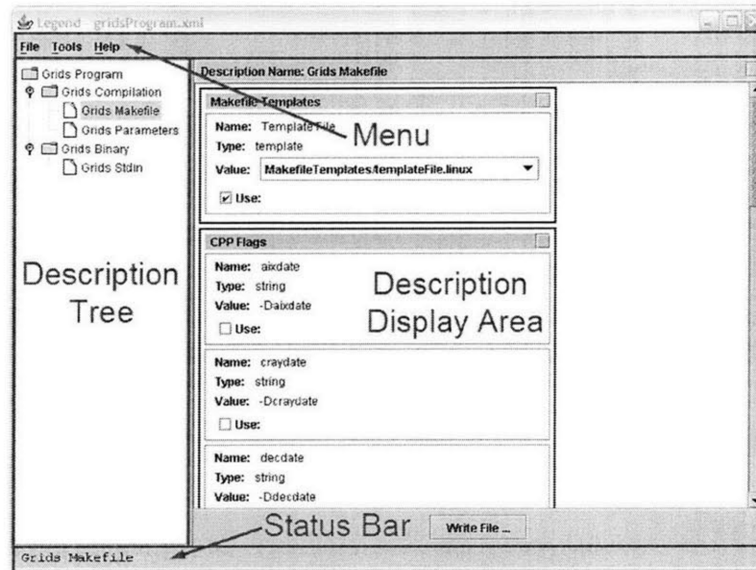


Figure 4-4: Example of opened LCML files with display areas labeled.

4.2.2 The LEGEND Menu

The LEGEND menu contains three submenus, the **File** menu, the **Tools** menu, and the **Help** menu.

The File Menu

The **File** menu contains the following choices:

- **Open ...** - Displays a dialog box used to open an LCML description file and its descendants, or select and open an exported state of a description and its descendants.
- **Open from web ...** - Similar to the **Open ...** menu option, except that this choice can be used to open a file from the internet.[4]

[4](i.e. a location that can be referenced via a URL (Uniform Resource Locator)).

- **Save XML Description ...** - Displays a dialog box used to save a complete copy of the LCML description that is currently being edited (with the current parameter values and settings).
- **Close** - Used to close the currently open LCML descriptions.
- **Import ...** - Displays a dialog box used to import a set of previously exported values to the currently selected description (and possibly its descendants).[5]
- **Export ...** - Displays a dialog box used to export an xml file containing values from the currently selected description (and possibly its descendants).[6]
- **Quit** - Used to exit the LEGEND program.

The Tools Menu

The **Tools** menu contains the following choices:

- **Refresh** - This option regenerates the user interface (re-displaying the current state of the description). When using a "table display" this can be used to return a sorted table to its initial order.
- **Extensions** - Holds a sub-menu which contains any plugins that are currently accessible from LEGEND. See Section 4.3.3 for additional information.
- **Options** - The options menu contains the following choices:
  - **No Schema Validation / Open with schema validation** - This option indicates whether LCML descriptions are to be validated against the LCML schema upon being opened. If **Open with schema validation** is selected, then LCML descriptions are expected to reference the LCML schema (and they will fail validation if they do not contain a reference to a schema).

[5]This feature is under developement.
[6]This feature is under developement.

- **Test Constraints On Action / Always Test Constraints** - This option specifies how often any constraints in the description are tested. If **Test Constraints On Action** is selected, constraints are only tested when file or script output is to be created from a description. The **Always Test Constraints** option is used to test constraints upon any edit of a variable's 'value' or 'use' setting as well as when file or script output is to be created.
- **Don't Show Hidden / Show/Edit Hidden** - This option controls whether "hidden" variables[7] are displayed to the user. If **Show/Edit Hidden** is selected, the "hidden" setting for variables is also shown and editable.
- **Use Panel Display / Use Table Display** - This option allows the user to choose between the default "panel display" and a "table display".
- **Export Current Description / Export From Current Down** - This option controls the export behavior. If **Export Current Description** is selected, exporting only stores the state of the description that is currently being edited. If **Export From Current Down** is selected, exporting stores the state of the description that is currently being edited and that of all its descendants. (If **Export From Current Down** is selected and the originally opened description file (the root of the description tree) is currently selected, exporting stores the state of the entire set of open descriptions.)

The Help Menu

The **Help** menu currently contains one item:

- **About ...** - Displays a dialog box that contains basic information about the program.

[7]i.e. LCML variables whose `hidden` element has the value 'true' (or whose reference resolves to the value 'true').

4.2.3 The LCML Description Tree

The LCML description tree contains the description names of all the open LCML descriptions. The root node of the description tree corresponds to the description that was originally "selected" to be opened (and if that description does not reference any "child descriptions" it will be the only node in the tree).

The tree is used to navigate through the LCML descriptions that are open. Selecting a description in the tree makes it the "currently selected" description, causing the description display area to show an interface corresponding to that description.

4.2.4 The LCML Description Display Area

The LCML description display area presents a graphical user interface for the currently selected description. It is automatically created by LEGEND and is used to configure the values and settings for the variables in the currently selected description. We will discuss various aspects of the description display area and its use.

Foldable Panel Display

The default interface for the description display area makes use of "foldable" panels.

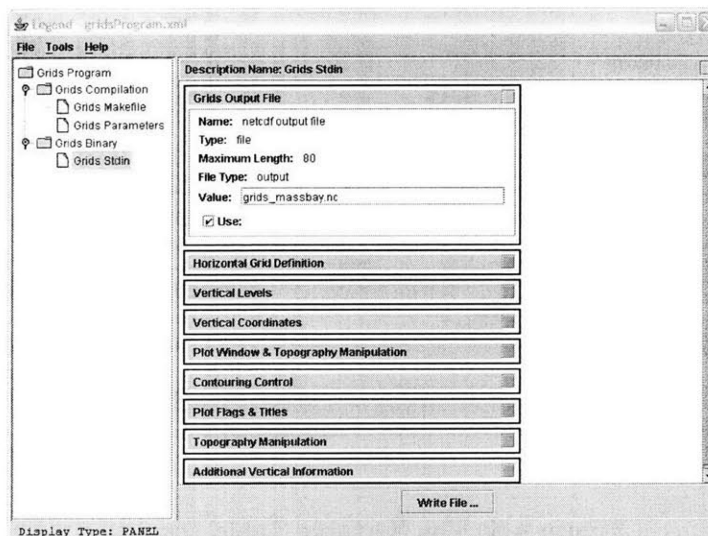


Figure 4-5: The Foldable Panel Display

Each panel corresponds to a "set" in LCML and can be collapsed (or restored) using the button at the panel's top-right corner. At the top-right corner of the entire description display area there is button that can be used to collapse all panels. When there is a large number of sets and variables in an LCML description, this arrangement facilitates the navigation to a specific set.

Sortable Table Display

An alternate interface for the description display area that makes use of a sortable table is available. The table display[8] provides two capabilities that are not present in the panel display:

- variables can be sorted (by set, name, value, etc.)
- all variable values are in one column (making it simpler and quicker to carry out a large number of edits.

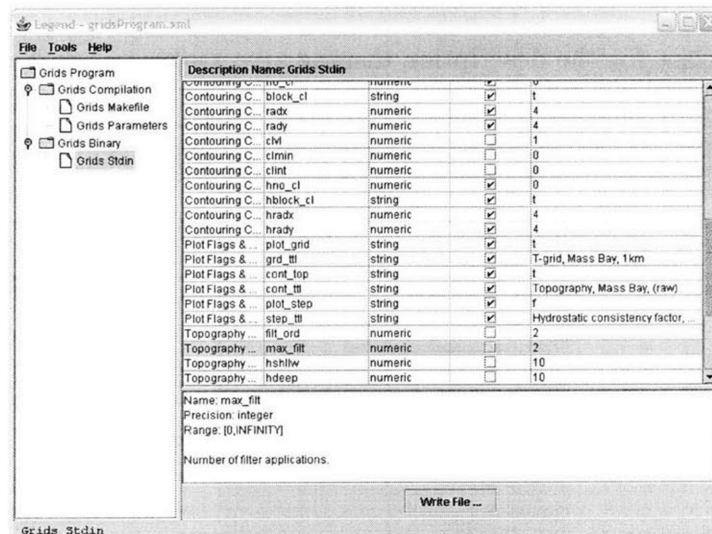


Figure 4-6: The Sortable Table Display

[8]This feature is under developement.

Display of Information

Information is contained in LCML descriptions at the *description*, *set*, *structure*, and *variable* levels. When the cursor hovers over a relevant area of LEGEND a piece of this information is displayed to the user as a "tooltip". For example, if the cursor hovers over the heading of a panel that displays a set, the information for that set should appear.

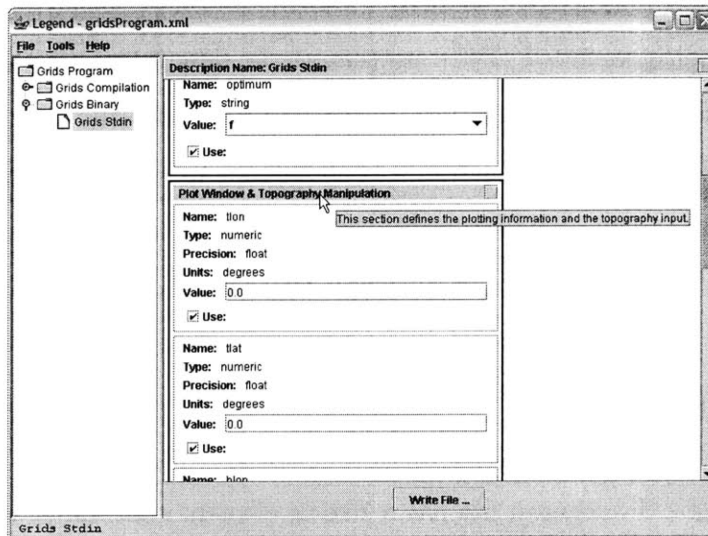


Figure 4-7: The Display of Set Information via a Tooltip

Editing

There are generally only two types of settings that are edited in an LCML description: the "use" and "value" settings of variables.[9] The general process of editing involves configuring the "value" and "use" settings for the variables in a description and then creating output (a file or a script) from the description.

Validation and Constraints

When the result of the editing of a variable's "value" does not pass validation an informational message is displayed to the user and the variable's value reverts back

[9]If the menu option **Show/Edit Hidden** is selected, a third setting, the "hidden" setting of variables is also editable

to its value prior to editing. When a constraint is violated a warning is simply displayed.[10]

Saving a File/Running a Script

If the currently selected description supports the creation of output to a file and/or a script then a panel is included at the bottom of the description display area for this purpose. A button is included for the creation of each relevant type of output (i.e. there may be a **Write File ...** button, a **Run Script ...** button, or both, depending on what variety of output the description supports).

Display of Multiplicity

It is possible that a description file has a `block` element containing variables with a multiplicity of values. Such a case is represented in the description display area with an **Edit Structure ...** button.

Selecting the **Edit Structure ...** button displays a dialog[11] that contains a table representing the content of the block's `structure` element. Each variable and child structure contained in the `structure` element is placed in its own column and the table contains a row for each `dataInstance` element of the structure.

If the block is multi-dimensional, the additional dimensions can be opened by clicking the **Edit Structure ...** buttons found in the table display of the original structure. Clicking on the heading of a variable's column displays details about that variable and provides access to its "use" setting (and possibly its "hidden" setting as well). The following figure displays the first and second dimension of a block:

[10]These functions are under development.
[11]This feature is under development.

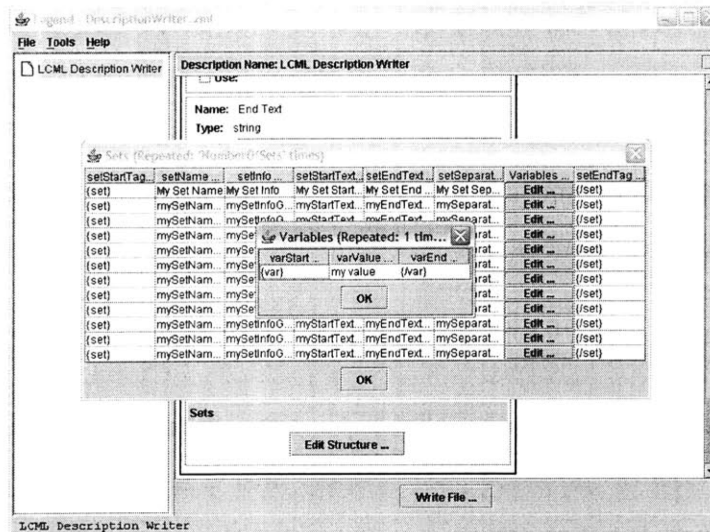


Figure 4-8: The Display of Multiple Structures

4.2.5 The Status Bar

The status bar is used to provide feedback to the user during program operation. In our implementation, it is typically used to present confirmation information that follows some sort of action.

4.3 Discussion of Design and Technical Considerations

4.3.1 User Interface Design and Implementation

General

Attention was paid to try to make the user interface simple to use and streamlined where possible. A general design principle used was to the minimize the number of mouse clicks that are necessary to complete basic actions. Also, attempts were made to follow standard conventions for user interface components. (For example, menu items and buttons that are used to display a dialog box end in the text ' ...').

Some of the features that were implemented to help the user easily edit descriptions include: relevant information from a description is displayed to the user in tooltips, variables that are uneditable are actually not editable, and variables with an enumeration are displayed using drop-down "combo-boxes" only permitting the selection of appropriate choices.

Folding Panel Display

As mentioned in Section 4.2.4, a user interface concept that was implemented in the "panel display" view of the description display area was the idea of *folding* panels. This idea of *folding* has seen use in integrated development environments and text editors. In the context of editing text (or code), folding allows portions of text to be "folded" or hidden from display and then later "unfolded" on demand. This can allow a user to work with a cleaner display and allow a user to navigate through a long display more quickly. A similar idea was also found in MAUI [18].

Our use of folding is similar, and in our application we are able to fold a "set". (When folded a set's contents are not displayed). A two-state button provides feedback to the user on the state of the folding. Also, a global folding button can be used to collapse all the sets. The global folding functionality does not unfold sets as it is unclear what action should be undertaken in the event that there are some sets that are folded and some that are not. In the interest of straight-forwardness the button's functionality is that it always collapses all sets. A normal (single-state) button is used for this case. The global folding functionality, in conjuction with the ability to fold single sets, allow a user a convenient manner by which to deal with only one set at a time (the user can simply fold all sets and then expand the ones of interest one at a time).

Sortable Table Display

While the default panel display of LCML description content provides a clear presentation of sets and their variables, it is not compact and is not well-suited for the quick traversal through and editing of large numbers of variables. The sortable table

display was developed to try to address these issues and to make it easier to find specific variables (again when there is a large number of variables).

The sortable table display contains the "table" at the top, and an "information panel" at the bottom. The information panel provides an alternative to the standard tooltip based information system in the event a user wants to read a long section of information more carefully, or in the (presumably) rare case that there is more information provided then will fit in a tooltip.

The development of the sortable table display involved extending the *JTable* Java Swing component. It was extended both to be able to display other components (labels, check boxes, and combo boxes) within it and to be sortable. The extension was based on the examples found in [1] and [27].

Remote File Browsing

Another issue that was addressed was what could be done to make it easier for a user to access a remote file than always having to manually enter an entire URL. The solution that was developed was the creation of a remote file browser.

The "remote file browser" was implemented using the HTML[12] rendering capabilities of the *JEditorPane* Java Swing component. While these capabilities do not match those of a full featured web browser, there is sufficient capability to allow a user to browse through most websites and thereby select a description file that is remotely available from a website. This approach works if the description files are linked to from a webpage or if remote browsing of the file system is supported for the remote folders of interest.

4.3.2 The Use of Runtime.exec()

If LEGEND is used to run scripts, it passes the value of the `command`[13] element from an LCML description to the 'Runtime.exec()' method in Java. If issues are encountered

[12] Hyper Text Markup Language

[13] The `command` element is a child of the `script` element, which is itself a child of the `descriptionTarget` element.

with the proper use of the `command` element that are not made clear by available examples, it is suggested that the reader consult the Java documentation and/or other resources related to 'Runtime.exec()' as there can be some intricacies involved in its proper use. In our development, examples from various online articles were considered so that we would properly handle the use and output of the 'Runtime.exec()' method.

4.3.3 Extension of LEGEND

As discussed previously one of the goals of this work is to support the integration of legacy programs with modern software development techniques. As a test case, we considered how LEGEND could be integrated with Sun Grid Engine (SGE) [11], a tool used for management of distributed computing.

Initially, a direct integration of SGE was planned. This would allow a typical LCML script to be run locally or using SGE. However, upon further consideration it was decided that it is preferable to describe how to launch a program into the SGE environment at the level of the LCML descriptions (this can be done since SGE is a command-line driven program). While this approach involves slightly more work for a description author, it is preferable, as it represents a far more flexible and general solution.

Some work had already been done on creating an "SGE Monitor" tool. This tool would be used for monitoring the status of SGE from LEGEND. It was then considered how this work could best be incorporated into LEGEND. Some LEGEND users might take advantage of such a tool for use with SGE, but many potential users may have no use for such a functionality. This could be true of many other potential tools that could be written for use with LEGEND (for example, someone who works with a specific types of data sets may might desire to make use of a visualization tool for them in conjunction with LEGEND). Given these considerations, it was concluded that a useful arrangement would be to allow "extensions" to LEGEND. In the envisioned setup, these extensions could be placed in a '.jar' file[14] and placed in the same directory as the LEGEND program. LEGEND would automatically detect

[14] A java archive file.

them and could add them to the **Extensions** submenu.[15,16]

4.3.4 Validation, Constraints, and References

When a user edits a variable's "value" in LEGEND, validation takes place as soon as a change is completed; this scheme identifies invalid variable values immediately. Note that the initial values of variables are never validated (as they should be correct).

With constraints LEGEND provides an option; constraints can be checked only upon the creation of output, or they can be checked upon any edits to a variable "value" or "use" setting as well as upon the creation of output. While the latter scheme indicates a violation of constraints whenever it occurs, there are cases where this could prove to be more of an annoyance to the user than a help. (Imagine a case where editing a single value results in the need to resolve a large number of constraints that must be dealt with individually).[17] If the former scheme (the default option) is selected the user can deal with any constraint violations at the time that output is created.

When a constraint violation occurs the text of the constraint's `explanation` element is displayed to the user in a warning message. LEGEND does not attempt to "fix" the constraint violation, and there is not sufficient information available to determine what should be fixed.[18]

It is also worth noting that the support of references (see Section 3.2.11) results in a rather dynamic situation. When a variable's "value" or "use" setting is updated, it should be tested if that "value" or "use" setting was referenced by any other element in the set of open LCML descriptions. If it is referenced by some other element than

[15]With such a structure the extensions could be independent of LEGEND with the exception of a method and field "signature" that allows them to recognized. They could even be java wrappers to external programs. Alternatively, they could make use of the LEGEND data model and classes, but they would still need to implement an appropriate method and field signature.

[16]This feature is currently under development.

[17]If there are ten constraints violated, and they are resolved one at a time, fixing the first one still results in the notification to the the user of nine constraint violations to be addressed, and the second, eight, and etc.

[18]For example, in the case of a conflict involving two variable values, it is not immediately clear which variable has the "wrong" value. (The conflict only expresses that a certain relationship between two values is not allowed).

the effect of this change should be considered. It may result in a validation event (if the referenced setting is used for another validated "value"), a constraint violation (if the more aggressive constraint checking scheme is used), or the need to update a multiplicity structure (if the referenced value serves as the dimension of a structure).[19]

4.3.5 Additional Details

The LEGEND software makes use of JDOM (Java Document Object Model) and the Apache Xerces-J software for accessing, parsing, and schema validating XML. The basis for the selection of these technologies and some additional details about them are discussed in [3].

LEGEND can be run as a Java application or as a Java applet. If it is run as an applet it should be bundled in a '.jar' file and signed. Details are again discussed in [3].

LEGEND is a work under development and is in the process of being made to support the complete LCML specification described in Chapter 3.

[19]The handling of the details that are discussed in this section is under development.

Chapter 5

Conclusions

5.1 Assessment of the Developed System

In this thesis we have discussed the development and use of a Legacy Computing Markup Langauge (LCML) and LEGEND, a processor for LCML. These two software technologies form an effective two-tier system for the encapsulation of command-line driven legacy programs.

A specification of the LCML language has been presented that is capable of handling a large number of situations that could appear in the encapsulation of a program. While LEGEND is still in the process of being made to support the complete LCML specification, basic functionalities have been implemented and they demonstrate the effectiveness of the approach. Most of the remaining functionalities have been successfully prototyped on previous "working" versions of the LCML specification.

An encapsulation of the Grids program of the HOPS software system was successfully undertaken. In addition, preliminary encapsulations of the remainder of the programs in the Harvard Ocean Predictions System were completed.

In its current state, the system can be used to run a program remotely by describing a third-party program (such as rsh, SGE, or Globus) that is capable of handling the remote command-line interactions and using the encapsulated program through the third-party program.

Lastly, it is worth noting that the development of LCML descriptions can be done

rapidly, and it is expected that use of the LCML/LEGEND system for encapsulating a command-line driven program provides a large savings in programming time and effort versus the development of a custom GUI.

5.2 Recommendations for Future Work

5.2.1 Basic Tasks

There are some basic tasks that require completion. In its current state the LCML specification has been further developed than the corresponding LEGEND implementation. LEGEND needs to be extended to support the complete LCML specification. Some particular areas that require additional work are the display of multiplicity, the use of references, and support for import/export.

Additionally, to complete our test case, the entire HOPS suite needs to be enscapsulated with LCML. Based on the results of our preliminary encapsulation of the remaining programs, it is expected that this will be pretty straightforward given the current LCML specification.

5.2.2 Integration of Client/Server Capability

While the current approach allows for a program to be run remotely by describing a third-party program to handle its remote use, another option is develop an integrated client/server capability into LEGEND that facilitates the remote execution of encapsulated programs. With such a capability, a user could edit descriptions using a client instance of LEGEND, and then use LEGEND to communicate with a remote server and run the encapsulated programs on the remote server.

Since the LEGEND program currently works as an applet, it is also possible that an applet-based client/server approach could be developed where LEGEND would be able to run programs on the server that the applet was downloaded from. In such a scenario, LEGEND could be used truly as a web-based interface. There are some implementation details that would need to be considered (basically how to

communicate with the server from the applet), as well as security issues that would need to be handled. Also, in its current state, LEGEND is a rather large program for widespread use as an applet. The possibility of triming down the libraries included in LEGEND to a minimum necessary functionality might be able to address this.

5.2.3 Integration with Globus

While the current approach allows integration with Globus [28] through the description of and use of the command-line, another option is a direct integration of Globus and LEGEND through the use of the Globus toolkit (and Java). Such an approach would "Grid" enable LEGEND, and in turn would "Grid" enable programs described in LCML.

5.2.4 Development of an LCML API

Currently, the LEGEND source code works with LCML descriptions via JDOM [15], which provides a representation of XML content in Java. A better approach may be to develop an LCML API that that models the structure of an LCML document in Java. Such an API would still make use of a technology like JDOM to create an LCML model from the underlying XML, but it would allow remainder of the LEGEND source code to deal with LCML objects instead of XML objects.

5.2.5 Handling of Units

Currently the `units` element (an optional element for numeric variables) is provided for information only. A more descriptive handling of units could be provided (for an example approach see [19]). Potentially, a system could be developed that allows LEGEND to be extended to allow for input in a variety of units (for example: kilometers, meters, and miles) and automatic conversion to the format expected for a certain program or file.

5.2.6 Enhanced Support of Mathematical Expressions

Currently, the validation of the "precision" and "range" of numeric variables can be bypassed by entering an equal sign '=' as the first character of the value of a numeric variable. This allows an expression to be entered in the place of an actual numeric value, as there are instances where this is supported by a program reading the file. However, it is not currently possible to validate the numeric value of such an expression. Instead of using the '=' flag to indicate the presence of an expression, the EVAL() function (see Section 3.2.11) can be used to provide a basic expression that can be evaluated.

A more flexible implementation might allow for the use of a more complicated mathematical expression in a variable's value, while still allowing for validation of the value of the expression. More complicated expressions could also be used in the specification of constraints. Their use, however, would require a standardized expression format and the incorporation of an expression parsing technology in LCML processing programs.

5.2.7 Formatted Input

We have not really considered the description of formatted input. By "formatted input" we mean input that has specific constraints on how it is to be entered. For example, a program may expect a variable to be specified in the form '`###.##`' or '`(###) ###-####`', (where the '`#`' indicates a digit should be present). One possible approach is to add an element to LCML that allows a "regular expression" to be provided for a variable and used for its validation.

5.2.8 User Interface Design and Usability of LCML

The design of the LEGEND user interface could be further considered. There is potential for adding right click menus and improved keyboard navigation (if these would represent an improvement). The current user interface could also be given some formal or informal usability testing to identify aspects that could be improved.

We could also test how well the LCML language is suited to the process of writing descriptions. This could be done by giving a small group of potential description authors a few sample programs to describe and gathering feedback on the process of description authoring. Such feedback could also be helpful in clearly formulating the needs of an LCML authoring tool, and identifying any needs for additional documentation of the language.

The work that has been done in investigating the development of an LCML authoring tool can also be continued, so that such a tool could be made available.

5.2.9 Display of Two-Dimensional Arrays

A special case of multiplicity is that of a two-dimensional array (represented by a structure containing only another structure, which contains only a single variable). In the current implementation of LEGEND such a case would be handled like any other instance of multiplicity and would be displayed as a one-dimensional array of one-dimensional arrays. However, for this special case it would be possible to implement a display that shows the full two-dimensional array in a single table.

5.2.10 Automation in LCML/LEGEND

Running or compiling a command-line driven program may involve multiple steps. A user may need to configure and save several different input files and then configure and launch a script. Currently, in such a case the user must know that each of these steps needs to be done, and there is no means to ensure that they are done. A possible development would be the implementation of a scheme by which the handling of these details could be partially automated, or by which a user could be guided through the process of configuring and operating such a program.

5.2.11 Automated Creation of LCML Descriptions

For some programs that have a "man" page (documentation that is readable with the Unix 'man' command), it may be possible to write a small program that reads

in the “man” page and outputs an LCML description. Such a technique is probably only plausible for programs with “man” pages that are sufficiently standardized, but in such cases it could represent a highly automated manner by which to create a graphical user interface for a command-line program.

Appendix A

Legacy Computing Markup Language (LCML) Roadmap

This section gives a terse overview of the LCML language. Optional elements are bracketed, elements with an asterix can occur more than once. Notes are indicated by numbers in parenthesis. If an element can contain a reference, it is marked with the text '(REF)'.

```
- description (1)
    - descriptionName
    - descriptionInfo
    - [descriptionMetadata]
        - [dc:title] (2)
        - [dc:creator] (2)
        - [dc:subject] (2)
        - [dc:description] (2)
        - [dc:publisher] (2)
        - [dc:contributor] (2)
        - [dc:date] (2)
        - [dc:type] (2)
        - [dc:format] (2)
```

    - [dc:identifier] (2)
    - [dc:source] (2)
    - [dc:language] (2)
    - [dc:relation] (2)
    - [dc:coverage] (2)
    - [dc:rights] (2)
- [descriptionChildren]
    - descriptionChild*
        - [location] (3),(REF)
        - [absoluteLocation] (3),(REF)
        - [description] (3)
        - [association]
- [descriptionTarget]
    - [file] (4),(REF)
    - [script] (4)
        - command (REF)
        - scriptname (REF)
- [descriptionContent]
    - [startText]
    - [endText]
    - [separator]
    - set*
        - setName
        - setInfo
        - [startText]
        - [endText]
        - [separator]
        - [var]* (5)
            - name
            - info

    - type
    - value (REF)
    - use (REF)
    - hidden (REF)
    - [header]
    - [trailer]
    - [enumeration]
    - [uneditable]
    - [aliases]
        - alias*
            - aliasKey (REF)
            - aliasOutput (REF)
    - [precision] (6)
    - [range] (6)
    - [units] (6)
    - [minLength] (6)
    - [maxLength] (6)
    - [multiLine] (6)
    - [fileType] (6)
    - [architecture] (6)
- [block]* (5)
    - structure
        - structureName
        - structureInfo
        - [startText]
        - [endText]
        - [separator]
        - numOccurs (REF)
        - [var]* (7)
        - [structure]* (7)

```
            - data
                - [dataInstance]* (8)
                    - [value]* (9),(REF)
                    - [data]* (9)
- [descriptionReplacements]
    - replacement*
        - replacementKey (REF)
        - replacementContent
- [descriptionConstraints]
    - [requirement]* (10)
        - explanation
        - condition
            - [test]* (11)
                - item (REF)
                - relation
                - item (REF)
            - [group]* (11)
                - [operator]
                - [test]* (12)
                - [group]* (12)
        - requires
            - [test]* (11)
            - [group]* (11)
    - [conflict]* (10)
        - explanation
        - [test]* (11)
        - [group]* (11)
```

Notes:

1. The `description` element can reference the LCML schema using `xmlns:xsi` and `xsi:noNamespaceSchemaLocation` attributes.
2. Any of the child elements of the `descriptionMetadata` element (the Dublin Core elements) can make use of the `xml:lang` attribute.
3. A `descriptionChild` element should contain either a `location`, an `absoluteLocation`, or a `description` element.
4. A `descriptionTarget` element should contain either a `file` or `script` element.
5. A `set` element should contain at least one `var` or `block` element as a child element and the occurences of these two elements can be intermingled.
6. This element is a "type dependent" element (only relevant for certain types of variables).
7. A `structure` element should contain at least one `var` or `structure` element as a child element and the occurences of these two elements can be intermingled.
8. The number of `dataInstance` elements should match the value of the `numOccurs` element for the corresponding structure.
9. The number and order of `value` and `data` elements should match the number and order of `var` and `structure` elements for the corresponding structure.
10. A `descriptionConstraints` element should contain at least one `requirement` or `conflict` element.
11. A `condition`, `requires`, or `conflict` element should contain at least one `test` element or one `group` element.
12. A `group` element should contain a total of at least two children that are either a `test` element or a `group` element.

Appendix B

LCML Description Schema

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

<!-- lcml.xsd -->
<!-- This schema should be used for LCML description documents. -->

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:dc="http://purl.org/dc/elements/1.1/">

<xs:import namespace="http://purl.org/dc/elements/1.1/"
           schemaLocation="simpledc20021212.xsd"/>

<xs:element name="description" type="descriptionItem"/>


<xs:complexType name="descriptionItem">
  <xs:sequence>
    <xs:element name="descriptionName" type="xs:string"/>
    <xs:element name="descriptionInfo" type="xs:string"/>
    <xs:element name="descriptionMetadata"
                type="descriptionMetadataItem" minOccurs="0"/>
```

```
    <xs:element name="descriptionChildren"
                type="descriptionChildrenItem" minOccurs="0"/>
    <xs:element name="descriptionTarget"
                type="descriptionTargetItem" minOccurs="0"/>
    <xs:element name="descriptionContent"
                type="descriptionContentItem" minOccurs="0"/>
    <xs:element name="descriptionReplacements"
                type="descriptionReplacementsItem" minOccurs="0"/>
    <xs:element name="descriptionConstraints"
                type="descriptionConstraintsItem" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>


<xs:complexType name="descriptionMetadataItem">
  <xs:sequence>
    <xs:group ref="dc:elementsGroup"/>
  </xs:sequence>
</xs:complexType>


<xs:complexType name="descriptionChildrenItem">
  <xs:sequence>
    <xs:element name="descriptionChild"
                type="descriptionChildItem" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>


<xs:complexType name="descriptionChildItem">
```

```
  <xs:sequence>
    <xs:choice>
      <xs:element name="location" type="xs:string"/>
      <xs:element name="absoluteLocation" type="xs:string"/>
      <xs:element name="description" type="descriptionItem"/>
    </xs:choice>
    <xs:element name="association" type="xs:string" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>


<xs:complexType name="descriptionTargetItem">
  <xs:choice>
    <xs:element name="file " type="xs:string"/>
    <xs:element name="script" type="scriptItem"/>
  </xs:choice>
</xs:complexType>


<xs:complexType name="scriptItem">
  <xs:sequence>
    <xs:element name="command" type="xs:string"/>
    <xs:element name="scriptname" type="xs:string"/>
  </xs:sequence>
</xs:complexType>


<xs:complexType name="descriptionContentItem">
  <xs:sequence>
    <xs:element name="startText" type="xs:string" minOccurs="0"/>
```

```
    <xs:element name="endText" type="xs:string" minOccurs="0"/>
    <xs:element name="separator" type="xs:string" minOccurs="0"/>
    <xs:element name="set" type="setItem" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>


<xs:complexType name="setItem">
  <xs:sequence>
    <xs:element name="setName" type="xs:string"/>
    <xs:element name="setInfo" type="xs:string"/>
    <xs:element name="startText" type="xs:string" minOccurs="0"/>
    <xs:element name="endText" type="xs:string" minOccurs="0"/>
    <xs:element name="separator" type="xs:string" minOccurs="0"/>
    <xs:choice maxOccurs="unbounded">
      <xs:element name="var" type="varItem"/>
      <xs:element name="block" type="blockItem"/>
    </xs:choice>
  </xs:sequence>
</xs:complexType>


<xs:complexType name="varItem">
  <xs:sequence>
    <xs:element name="name" type="xs:string"/>
    <xs:element name="info" type="xs:string"/>
    <xs:element name="type" type="xs:string"/>
    <xs:element name="value" type="xs:string"/>
    <xs:element name="use" type="xs:string"/>
    <xs:element name="hidden" type="xs:string"/>
```

```
    <xs:element name="header" type="xs:string" minOccurs="0"/>
    <xs:element name="trailer" type="xs:string" minOccurs="0"/>
    <xs:element name="enumeration" type="xs:string" minOccurs="0"/>
    <xs:element name="uneditable" type="xs:string" minOccurs="0"/>
    <xs:element name="aliases" type="aliasesItem" minOccurs="0"/>

    <!-- The remaining elements in the "var" element are
         type dependent and should only be used in conjuction
         with an appropriate value of the "type" element.
         This is not checked by the schema. -->

    <xs:element name="precision" type="xs:string" minOccurs="0"/>
    <xs:element name="range" type="xs:string" minOccurs="0"/>
    <xs:element name="units" type="xs:string" minOccurs="0"/>
    <xs:element name="minLength" type="xs:string" minOccurs="0"/>
    <xs:element name="maxLength" type="xs:string" minOccurs="0"/>
    <xs:element name="multiLine" type="xs:string" minOccurs="0"/>
    <xs:element name="maxLength" type="xs:string" minOccurs="0"/>
    <xs:element name="fileType" type="xs:string" minOccurs="0"/>
    <xs:element name="architecture" type="xs:string" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>


<xs:complexType name="aliasesItem">
  <xs:sequence>
    <xs:element name="alias" type="aliasItem"
                minOccurs="1" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
```

```
<xs:complexType name="aliasItem">
  <xs:sequence>
    <xs:element name="aliasKey" type="xs:string"/>
    <xs:element name="aliasOutput" type="xs:string"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="blockItem">
  <xs:sequence>
    <xs:element name="structure" type="structureItem"/>
    <xs:element name="data" type="dataItem"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="structureItem">
  <xs:sequence>
    <xs:element name="structureName" type="xs:string"/>
    <xs:element name="structureInfo" type="xs:string"/>
    <xs:element name="startText" type="xs:string" minOccurs="0"/>
    <xs:element name="endText" type="xs:string" minOccurs="0"/>
    <xs:element name="separator" type="xs:string" minOccurs="0"/>
    <xs:element name="numOccurs" type="xs:string"/>
    <xs:choice maxOccurs="unbounded">
      <xs:element name="var" type="varItem"/>
      <xs:element name="structure" type="structureItem"/>
    </xs:choice>
```

```
  </xs:sequence>
</xs:complexType>


<xs:complexType name="dataItem">
  <xs:sequence>

    <!-- The number of dataInstances should match the value
         of the "numOccurs" element of the appropriate
         structure. This is not checked by the schema. -->

    <xs:element name="dataInstance" type="dataInstanceItem"
                minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>


<xs:complexType name="dataInstanceItem">
  <xs:sequence>

    <!-- The number of and order of "value" and "data"
         elements should match the number and order of
         "var" and "structure" elements for the appropriate
         structure. This is not checked by the schema. -->

    <xs:choice maxOccurs="unbounded">
      <xs:element name="value" type="xs:string"/>
      <xs:element name="data" type="dataItem"/>
    </xs:choice>
  </xs:sequence>
```

```
</xs:complexType>


<xs:complexType name="descriptionReplacementsItem">
  <xs:sequence>
    <xs:element name="replacement"
                type="replacementsItem" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>


<xs:complexType name="replacementsItem">
  <xs:sequence>
    <xs:element name="replacementKey" type="xs:string"/>
    <xs:element name="replacementContent"
                type="descriptionContentItem"/>
  </xs:sequence>
</xs:complexType>


<xs:complexType name="descriptionConstraintsItem">
  <xs:choice maxOccurs="unbounded">
    <xs:element name="requirement" type="requirementItem"/>
    <xs:element name="conflict" type="conflictItem"/>
  </xs:choice>
</xs:complexType>


<xs:complexType name="requirementItem">
  <xs:sequence>
```

```
    <xs:element name="explanation" type="xs:string"/>
    <xs:element name="condition" type="conditionItem"/>
    <xs:element name="requires" type="requiresItem"/>
  </xs:sequence>
</xs:complexType>


<xs:complexType name="conditionItem">
  <xs:choice maxOccurs="unbounded">
    <xs:element name="test" type="testItem"/>
    <xs:element name="group" type="groupItem"/>
  </xs:choice>
</xs:complexType>


<xs:complexType name="requiresItem">
  <xs:choice maxOccurs="unbounded">
    <xs:element name="test" type="testItem"/>
    <xs:element name="group" type="groupItem"/>
  </xs:choice>
</xs:complexType>


<xs:complexType name="conflictItem">
  <xs:sequence>
    <xs:element name="explanation" type="xs:string"/>
    <xs:choice maxOccurs="unbounded">
      <xs:element name="test" type="testItem"/>
      <xs:element name="group" type="groupItem"/>
    </xs:choice>
```

```
  </xs:sequence>
</xs:complexType>


<xs:complexType name="testItem">
  <xs:sequence>
    <xs:element name="item" type="xs:string"/>
    <xs:element name="relation" type="relationItem"/>
    <xs:element name="item" type="xs:string"/>
  </xs:sequence>
</xs:complexType>


<xs:complexType name="groupItem">
  <xs:sequence>
    <xs:element name="operator" type="operatorItem" minOccurs="0"/>
    <xs:choice minOccurs="2" maxOccurs="unbounded">
      <xs:element name="test" type="testItem"/>
      <xs:element name="group" type="groupItem"/>
    </xs:choice>
  </xs:sequence>
</xs:complexType>


<xs:simpleType name="relationItem">
  <xs:restriction base="xs:string">
    <xs:enumeration value="SAME"/>
    <xs:enumeration value="DIFF"/>
    <xs:enumeration value="EQ"/>
    <xs:enumeration value="NEQ"/>
```

```
    <xs:enumeration value="GT"/>
    <xs:enumeration value="GEQ"/>
    <xs:enumeration value="LT"/>
    <xs:enumeration value="LEQ"/>
  </xs:restriction>
</xs:simpleType>


<xs:simpleType name="operatorItem">
  <xs:restriction base="xs:string">
    <xs:enumeration value="AND"/>
    <xs:enumeration value="OR"/>
    <xs:enumeration value="XOR"/>
  </xs:restriction>
</xs:simpleType>


</xs:schema>
```

Bibliography

[1] Z. Anjum. Display any JComponent in a cell. http://www.codeguru.com/java/articles/162.shtml.

[2] Y. Bi, M. E. C. Hull, and P. N. Nicholl. An XML approach for legacy code reuse. *The Journal of Systems and Software*, 61(2):77–89, 2002.

[3] R. Chang. The encapsulation of legacy binaries using an XML-based approach with application in ocean engineering. Master's thesis, Massachusetts Institute of Technology, May 2003.

[4] Condor-G. http://www.cs.wisc.edu/condor/condorg/.

[5] Document structure description. http://www.brics.dk/DSD/.

[6] Dublin core metadata initiative. http://www.dublincore.org.

[7] C. Evangelinos, R. C. Chang, P. F. J. Lermusiaux, and N. M. Patrikalakis. Web-enabled configuration and control of legacy codes: An application to ocean modeling. *International Journal of Cooperative Information Systems*, 2004. To appear.

[8] Extensible markup language website. http://www.w3.org/XML/.

[9] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the Grid: Enabling scalable virtual organizations. *International Journal of Supercomputing Applications*, 15(3), 2001.

[10] Global Grid Forum. http://www.ggf.org.

[11] Grid Engine. http://gridengine.sunsource.net/.

[12] Harvard Ocean Prediction System (HOPS). http://oceans.deas.harvard.edu/HOPS/.

[13] Java foundation classes (JFC/Swing). http://java.sun.com/products/jfc/.

[14] Java technology website. http://java.sun.com/.

[15] JDOM. http://www.jdom.org/.

[16] jEdit - open source programmer's text editor. http://www.jedit.org/.

[17] Catherine Letondal. PISE, a tool to generate web interfaces for molecular biology programs. http://www.pasteur.fr/recherche/unites/sis/Pise/.

[18] MAUI. http://csmr.ca.sandia.gov/projects/maui/.

[19] F. Olken and J. McCarthy. Measurement units in XML datatypes. http://pueblo.lbl.gov/~olken/mendel/w3c/xml.schema.wg/units/syntax.htm.

[20] OpenOffice.org. http://www.openoffice.org/.

[21] N. M. Patrikalakis, J. J. McCarthy, A.R. Robinson, H. Schmidt, C. Evangelinos, P. J. Haley, S. Lalis, P. F. J. Lermusiaux, R. Tian, W. G. Leslie, and W. Cho. Towards a dynamic data driven system for rapid adaptive interdisciplinary ocean forecasting. In F. Darema, editor, *Dynamic Data-Driven Application Systems*. Kluwer Academic Publishers, Amsterdam, 2004. To appear.

[22] C. Phanouriou and M. Abrams. Transforming command-line driven systems to web applications. In *Selected Papers from the Sixth International Conference on World Wide Web*, pages 1497–1505. Elsevier Science Publishers Ltd., 1997.

[23] Pollo. http://pollo.sourceforge.net/.

[24] Poseidon - rapid real-time interdisciplinary ocean forecasting: Adaptive sampling and adaptive modeling in a distributed environment. http://czms.mit.edu.

[25] RELAX NG home page. http://www.relaxng.org/.

[26] SGML/XML: Using elements and attributes. http://www.oasis-open.org/cover/elementsAndAttrs.html.

[27] N. Tamemasa. Sorting the table. http://www.codeguru.com/java/articles/219.shtml.

[28] The Globus alliance. http://www.globus.org/.

[29] E. Wohlstadter, S. Jackson, and P. Devanbu. Generating wrappers for command line programs: the Cal-Aggie Wrap-O-Matic project. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 243–252. IEEE Computer Society, 2001.

[30] XML schema. http://www.w3.org/XML/Schema.

[31] XML schema tutorial. http://www.w3schools.com/schema/.