Development and Use of Tools for Modularization of Activation Programs

Milad Fatenejad

December 2005

UWFDM-1289

FUSION TECHNOLOGY INSTITUTE

UNIVERSITY OF WISCONSIN

MADISON WISCONSIN

Development and Use of Tools for Modularization of Activation Programs

Milad Fatenejad

Fusion Technology Institute
University of Wisconsin
1500 Engineering Drive
Madison, WI 53706

http://fti.neep.wisc.edu

December 2005

UWFDM-1289

Development and Use of Tools for Modularization of Activation Programs

by:

Milad Fatenejad

A thesis submitted in partial fulfillment of the

requirements for the distinction of

Honors in Research

(Nuclear Engineering and Engineering Physics)

at the

UNIVERSITY OF WISCONSIN – MADISON

2005

1 INTRODUCTION

Activation codes are important tools that can be used to determine the potential safety risks and waste disposal ratings associated with materials that are exposed to radiation. Traditionally, many of these codes are developed in a monolithic way that greatly increases the size of the applications, and decreases the productivity of the scientists and engineers involved in their development.

In computer science, modular programming involves breaking a code into multiple pieces that communicate with one-another. Each module is independent of the others, and communication occurs through well-defined interfaces. In the case of an activation code, several general parts can easily be identified.

1. Code that performs input/output
2. Code that retrieves nuclear data
3. Code that performs the actual activation calculation

Under the present system, every activation code implements its own routines to perform these actions. Because of this, a great deal of development time is spent on tasks that do not directly relate to the actual physics parts of the program that calculate the isotopic content of materials. Some of these tasks included parsing input files or loading often-cryptic nuclear data files into memory.

The goal of this project was to create tools and guidelines that can be used to modularize activation codes. The project was divided into three sections, each focusing on one of the points listed above. Modularization will provide many important and practical benefits for scientist and engineers that write activation codes. Most importantly, modularization will in one form or another allow scientists

to focus on the physics parts of their applications, while reducing the volume of unassociated code.

1.1 Project Overview

This project focused on three sections, which involved input, accessing nuclear data, and finally, performing the activation calculation itself. All activation codes require all three of these components, and it is possible to use external libraries and generalized programming methods to reduce the amount of effort it takes to implement each component.

The input section of the project focused on using third party XML libraries to perform the task of actually parsing the input files. These libraries can also perform some rudimentary error checking, that can greatly improve the usability of programs. A large amount of code can be off-loaded to the XML library, which leads to many benefits.

To perform nuclear data access, a tool called the Fast Easy Interface to Nuclear Data (FEIND) was developed. This library can be linked to activation programs and can load and store nuclear data so that code developers can focus on more important parts of their applications. This helps reduce redundancy between activation codes which must each implement methods for parsing nuclear data files.

Finally, a small activation code, AlCore, was developed. This code is designed to allow the addition of multiple methods for solving the activation problem in an easy and modular way. It is also extremely small, and can easily be linked to other programs that may desire quick access to activation capabilities.

2 XML INPUT

2.1 Motivation and Goals

2.1.1 The Current Situation

Reading and parsing input files is a task common to many scientific computing applications. The code developer usually designs some syntax for the input file, and then writes code that is responsible for reading that file format and storing its contents in memory. As this task is completed, good developers will also perform various levels of error checking, ranging from checking the syntax of the file to verifying the validity and self-consistency of its content.

The amount of code required to perform all of these tasks can be substantial. For example, ALARA [1], a program designed and run at the University of Wisconsin, uses nearly half of its total code base to parse input data. Such a large volume of code can introduce bugs into the application. Furthermore, a great deal of time, particularly during the initial phases of development is spent writing input related routines, which can be a distraction for scientists and engineers.

Furthermore, the file syntax may be difficult to use, because many programmers involved in scientific computing may not have a great deal of interest in designing input file formats. The end result often times is that developers create input files that are easy for them to parse, but is somewhat more difficult for users understand.

2.1.2 Goals

XML libraries can be used to solve many of these problems. XML is a standard

language that is used to describe data, and there are many free XML libraries that can be used to parse XML files. The goal of this project was to develop a method for adding XML support to activation programs. To demonstrate these methods, such support was added to a new Monte Carlo activation analysis code development project underway at the University of Wisconsin. XML was used to improve the code, by providing error free input, and also improved the user experience by creating an easy to use input file format that is readily understandable. Prior to this work, the code did not have an input file.

2.2 XML Introduction

XML is a markup language that was designed as a tool to describe data [2]. It is not a programming language, in the sense that XML does not actually give commands to the computer. It is merely a language that can be used to store data in an easy to read manner. An XML file can be used as an input file and can store the data an activation program needs to run.

Unlike “home-made” input file formats, XML files are not read directly by the activation code. Instead, an XML parsing library is responsible for reading the file. Many of these libraries exist, and many of the highest quality ones are free to use [3][4]. The user application links to these libraries, and can access the data through them.

When the libraries read in the data, they perform a step known as validation on the XML file. During this process, they compare the input file to a separate grammar file which defines what the different elements and objects in the input file are, and in what order they should occur. The library ensures that the input file conforms to the

grammar and will display an error if it does not. Validation is performed by the library and provides a great deal of automatic error checking. The end result it that the activation code developer does not need to perform this error checking himself, and the total volume of the associated code is reduced.

2.2.1 XML Advantages

There are several advantages provided by using XML files over plain text files. All of these are important because they reduce the amount of work a developer must do to process input files, and allows someone else to work on the problem. It allows for a division of labor among various experts, which is one of the central benefits provided by modularization. Scientists and engineers can focus on math and physics, while computer scientists can focus on writing better parsers.

Furthermore, XML style syntax is something that is familiar to a large number of people in general computing. It is very strict and very standardized making it easy to use. The language is verbose which forces users to create input files that are readily understandable to human readers as well as the software application.

The importance of XML validation must also be stressed at this point. The fact that XML relies on a separate file for a grammar definition is very beneficial, because it means that developers simply modify the schema or DTD instead of having to resort to FORTRAN, C++, or other low level languages. Performing all of the validation that XML libraries can perform for you would require a very large block of code, as opposed to a relatively small schema. Verifying the function of the schema or DTD also does not require recompiling the user application.

2.2.2 XML Syntax

XML files are designed to be easy to use. The World Wide Web Consortium (W3C) is the body that is responsible for maintaining the XML standard [5], and adds features to the language as they become necessary. XML looks very similar to the more familiar HTML, but there are important differences. Shown below is an example of an XML file that is used to describe a note [2].

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<note>
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
</note>
```

The file should be fairly easy to understand by simply glancing at it. It is clear that there is a section describing who the note is from, who it is addressed to, what the subject is, and finally what the body of the note contains.

XML files are comprised of a series of *elements.* Elements are surrounded by a set of *tags,* which are surrounded by angle brackets. These tags indicate where elements begin, and where they end. For example,

```
<to>Tove</to>
```

is a "to" element which contains the word "Tove." Elements may contain other elements, making the XML files hierarchical in nature. The first element in a file is called the *root* element. In the above example, the root element is "note." The file also contains elements called "to", "from", "heading" and "body." The file is easily understandable which will make XML generally beneficial to programs needing input files.

The first line is the example clearly states what version of the XML standard the file uses. This helps activation code users by preventing confusion due to the XML standard itself changing. This happens fairly infrequently, and the changes are usually not dramatic.

2.2.3 XML Parsing Libraries

Many XML libraries have been written that are freely available. Each has its own benefits and drawbacks. Initially, libxml++ was considered which essentially provides C++ wrapper functions for the popular libxml [3] library. However, this library was soon abandoned due to a lack of full validation support [6].

Eventually, it was decided that the Xerces XML library [4] was appropriate for this project. Xerces is a popular XML library that is written by the Apache Software Foundation. Although it has a somewhat more cumbersome interface that libxml++, it does support full validation (DTDs and schemas).

The library also conforms to the W3C Document Object Model (DOM) Binding specification [7]. This means that the library's interface to the activation application is standardized. This is beneficial for activation code developers, because once the DOM binding is learned, it is easy to implement XML support in other programming languages. For example, if one wanted to write a front end for an activation application using another language such as python, which also conforms to the DOM specification, much of the knowledge gained using Xerces could be reused, even though a completely different library might be involved.

2.2.4 Validation

Validation in XML is a process in which the parser verifies that the XML input file contains the correct elements in the correct order by comparing it to an XML *schema* or *DTD*. These are the grammar files which define how the XML input file must look. For the example shown in section 2.2.2, a corresponding XML schema looks similar to the code shown below. This is not a complete schema, but effectively illustrates the point [2].

```
<xs:element name=”note”>

    <xs:complexType>
      <xs:sequence>
        <xs:element name="to" type="xs:string"/>
        <xs:element name="from" type="xs:string"/>
        <xs:element name="heading" type="xs:string"/>
        <xs:element name="body" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
</xs:element>
```

This example schema tells the XML parser that the root element is note, and that note must contain a sequence of elements. These elements are “to”, “from”, “heading”, and “body”, and they must appear once each and in that order. If, during validation, the XML input file does not correctly match the schema or DTD, the parser will usually either exit or throw an exception.

Basic tasks that may be performed in validation involve checking the data types of elements, and more importantly, defining which elements should occur in which order and how many times they should occur. Validation by no means provides all of the error checking that is required in activation calculations, but does significantly reduce the total volume of code that is required. During initial stages of application

development, error checking input files can be very time consuming, and in some cases, difficult process. Writing a schema or DTD is much easier than incorporating the error checking code directly into the application. Furthermore, making changes to the schema or DTD does *not* necessarily require making any changes to the code.

Once it is recognized that validation is an extremely beneficial part of using XML, it is necessary to chose whether to implement it using either a schema or DTD. Each validation system has its own benefits. Both grammars are stored in external text files, and in both cases, the parser will automatically perform the validation. The DTD represents the older of the two validation systems. It is generally regarded as being inferior to a schema, and it was because of its limitations that the schema was developed and added to the XML standard.

Schemas are superior to DTDs in several ways. First, they support all of the more advanced features of XML, such as namespaces [8]. Namespaces allow XML users to assign their elements unique identifiers that will prevent multiple uses of common element names. Furthermore, schemas are themselves XML files. This is very important because it allows the definition of a grammar without having to learn an entirely new language. This is unlike DTDs which use a different syntax.

Most importantly, schemas allow users to more clearly specify what the contents of the XML file should be. It has a system in place for defining what data types elements should contain. For example, users can specify that a particular element should contain a generic string, a time, or a non-negative integer. Definitions of custom data types are also allowed. For this project, data types for representing isotopic identifiers, and non-negative floating point numbers were needed. Therefore,

XML schemas allow for basic types of validation in addition to type checking, and advanced feature support, all of which will help improve the underlying activation application.

2.3 NorMC

As a proof of principal, an activation code that is being developed at University of Wisconsin, which for lack of an actual name will be referred to as NorMC after its primary developer, was modified to use XML style input. Not only would this demonstrate the amount of error checking and ease with which XML support could be added to an application, but a system was developed for doing this as unobtrusively as possible. Furthermore, it demonstrated that in many cases it is possible to add support to an application with little to no knowledge of how the program actually works. I personally was not at all familiar with the inner workings of NorMC, but thought it important to show that it was not necessary, because it demonstrates that it is possible for code developers to out-source some of this work to other programmers, if desired.

Some background into how NorMC works is appropriate. It is a Monte Carlo style activation program. The user specifies various control volumes which contain different fluxes. The control volumes are linked together, and the total amount of time particles should spend in the control volumes are also indicated. These volumes are linked to one another and probabilities of traveling between them are given. The user then defines the various source particles that begin at different times at different control volumes, and slowly work their way through the problem, until some end condition is reached. If enough particles are run, the program will converge on the

correct isotopic abundances.

2.3.1 XML Input Design

The XML input module for NorMC was required to meet several goals. The code itself was written in an object oriented style in the C++ programming language. Shown below are several of the requirements for the input module.

- Must be Non-intrusive
- Must require a low NorMC learning curve
- Must perform complete validation of the data

The first of these requirements, non-intrusiveness, is important to prevent serious disruption in the rest of the code's development. Ideally, the module would be as self contained as possible, and would have a very simple interface to the rest of the program. The second, requiring a low learning curve, simply illustrates the fact that I am not very familiar with NorMC, and that the implementation must not require a strong familiarity. This is also important because ideally, implementing XML input in most activation applications can be done in such a way so as not to require much expertise. Good general design in NorMC helped facilitate this requirement.

The final requirement is that the module perform complete validation of the data. This involves more than simply defining a schema. There are other errors that may arise that must be dealt with separately. The goal is that the separate input module will completely verify the integrity of the data for the physics part of NorMC. This can help reduce the size of the physics module by removing some error checking from that part, thereby making it more readable.

The layout of the program is shown below, in figure 2.1. Information from the

input file and the schema is first read in by the XML parser, which performs the first level of error checking – validation against a schema. The data is then sent to the NorMC input handler which will perform additional error checking on it before sending it to the physics part of the code. The input handler fills in various data structures that have already been set up in the program.

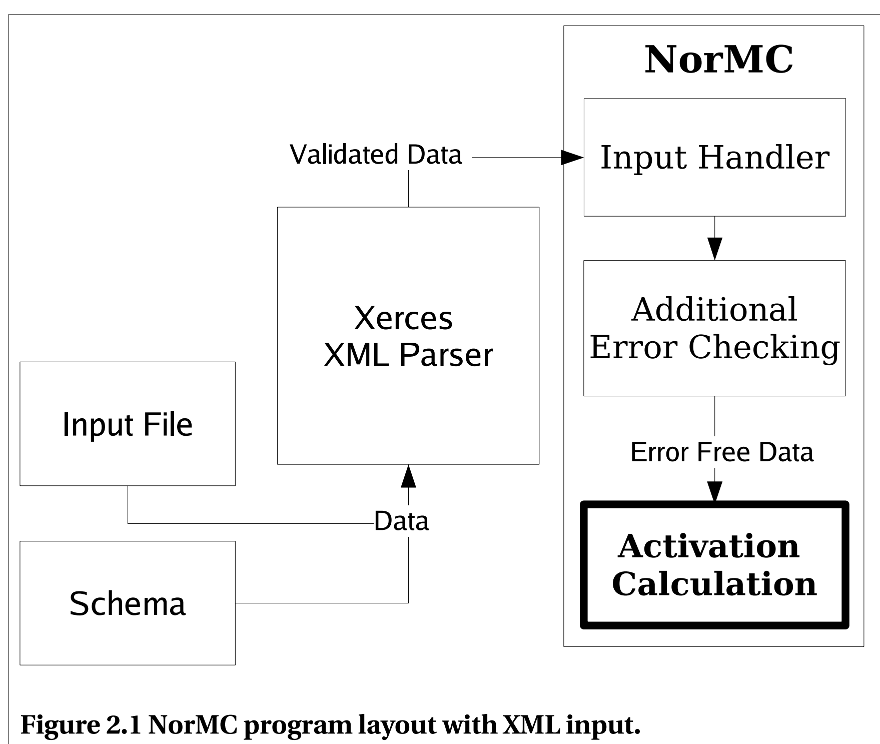

Figure 2.1 NorMC program layout with XML input.

As was mentioned in 2.2.4, a schema is primarily used to define the order and behavior of various data *types*. It does not provide many restrictions on the data *content*. For this reason, additional error checking is needed. An example of this is provided later in this report.

2.3.2 Input Handler

The input handler receives validated data from the XML parser and is expected to prepare the data so the physics part of the code can use it. There was some discussion about how this section of the program should be written.

The first option represents a more procedural approach. The input routines would ask the parser for certain elements in a certain order. As the elements were read, the program would methodically read each elements' children in a specified order. The benefit of this method is that more flexibility is allowed in the input file. However, an important drawback is that a great deal of information about the structure of the input file is built into the code, making the code less flexible to changes in the schema.

To understand what is meant by this, it is beneficial to look at the second option. The idea was to use a recursive function to simply load each element in what ever order they might appear. Each element has some function associated with it that will be called if that element is encountered. To simplify this process as much as possible, additional constraints can be added to the input file using the schema. This method requires less code, and less knowledge of how the input file is set up. However, the input file becomes more restrictive.

Finally, a third option exists, which uses a recursive system similar to second option. In this case, however, the input file is not restricted at all. To compensate for this, a great deal of code is added to make up for the input file limitations. The benefits that are realized in terms of code readability and modularization are over shadowed by the large volume of additional code that is needed. Because of this, the third option is not desirable.

The final implementation used the second option, because it represented a more generalized method for implementing XML input in activation codes, and took advantage of the schema file to a larger degree. In other words, the purpose of using

XML input was to externalize as much of the programming as possible to the XML library and schema, and the second option represented a good balance between ease of use on the part of developers and users.

It should be noted that while the input file became more rigid, it by no means became difficult to use. It simply established an order in which certain elements must appear. Furthermore, one of the strengths of the many XML libraries is the quality of error messages. In the case of the Xerces library, if certain elements are placed out of order, a message is printed specifying the offending element and the exact order in which the elements should appear. This can help reduce the need to resort to large manuals to determine the correct order of elements.

2.3.3 The NorMC input file

This section will introduce an example NorMC XML input file, figure 2.2, and will describe the various validation techniques, both through the schema and not, that are used to provide error free input to the code's physics components. Appendix A contains the full XML schema.

The Monte Carlo calculation represented by the input file shown in figure 2.2, will run one million particles for a total simulation time of one year and uses FENDL decay and transmutation data. There is only a single source consisting of Iron-56 isotopes, and a single control volume.

The XML schema puts several constraints on how the input file must be written that reduces the size of the input handler, as was discussed in the previous section. For example, the FluxDef element defines an energy dependent group-wise neutron flux. Multiple flux definitions are allowed in a problem that can be used by many

control volumes. The ControlVolume element contains a flux element which specifies which of the previously defined fluxes exist in that ControlVolume. In this case, the control volume named “start” contains a neutron flux named “flux1”. The schema is used, in this case, to force the definition of the FluxDef elements before the ControlVolumes. Because of this, when the input handler processes a ControlVolume

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

<DMCIA
xmlns="DMCIA"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="DMCIA ../../src/input/DMCIA.xsd">

<NPS num="1000000" />
<TotalTime time="315360000"/>

<!-- DATA LIBRARY PATHS -->
<TransLib path="/home/fateneja/nuclear-data/fendlg-2.0_175" />
<DecayLib path="/home/fateneja/nuclear-data/fendld-2.0" />

<FluxDef name="flux1">
  <Point value="0.0" repeat="7" />
  <Point value="5.0E-12" repeat="168" />
</FluxDef>

<AtomCurrentTally name="ac1" time="315360000.0"/>

<ControlVolume name="start" res_time="315360000.0" >
  <Flux name="flux1" />
  <Tally name="ac1" />
</ControlVolume>

<Mixture name="iron">
  <Atom kza="260560" amount="91.75"/>
</Mixture>

<!-- DEFINE FE-56 SOURCE -->
<Source name="source1" mix="iron" cv="start" time="0.0" strength="2"/>

</DMCIA>
```

Figure 2.2 Simple NorMC input file

element which contains a neutron flux, it does not have to prepare for the possibility that the flux is defined later in the file. If the flux has not been defined at that point, it

is certainly an error.

This case also provides a good example of auxiliary error checking. Earlier, it was mentioned that this was needed to identify potential problems with the content of elements. In this case we are verifying that the flux requested in the ControlVolume matches one of the fluxes that has previously been defined.

2.4 Conclusion

The attempt to add XML support to NorMC was very successful. The code did not previously have any means of reading input, and can now rely on validated error free input being provided by the input handler. These changes are now a part of the mainline development of the code. The XML additions were made to NorMC with very little knowledge of how the code functioned, and was done in a non-intrusive way. This project demonstrated the feasibility and benefits associated with using XML as a means of providing input to activation codes.

3 FEIND

3.1 Motivation and Goals

Accessing and storing nuclear data is an important part of activation codes. Nuclear data is created by various groups which are responsible for maintaining the data and verifying its integrity. Once data is created, it is retrieved by activation code developers who are responsible for adding support for it within their codes. There are several problems with the current system for creating and storing data; both for format maintainers and code developers.

Data is usually created in many different formats which define how it is stored within a file. They do not govern the actual content of the file. In other words identical data can be stored in files with two different formats. Each data format requires specialized code to parse it. Parsers are blocks of code that are responsible for loading data from files into memory and are unique to the individual data formats. Currently, developers for each activation code are responsible for writing their own parsers. This situation creates a great deal of cross-application redundancy and is not dissimilar to one of the motivations for XML input (see section 2.1). Developers are forced to divert attention away from writing numerical methods and physics parts of their codes in order to perform the relatively tedious process of creating a parser. This occurs not only when codes need access to a new format, but also when the data formats themselves change.

Redundancy is generally regarded in a negative light for other reasons. The more implementations of identical code that exist, the more likely one or more of the implementations can contain serious errors. Errors in data handling routines not only

introduce inaccuracy in codes, but again divert additional attention away from more important aspects of programs.

Because activation code developers are responsible for writing their own parsers and data storage structures, maintainers of data formats also suffer. Making changes to data formats can becomes a very difficult process. The data formats may need backwards compatibility to ensure that activation codes can use the most up to date data without breaking functionality. Furthermore, the formats need to be written in a way that is relatively understandable by humans, which can result in slow processing time for computers. For this reason many formats are stored in human readable text files, which are usually slow to process and require a large amount of storage space.

FEIND is a tool that will address many of these issues by acting as an intermediary between data format maintainers and activation code developers. Activation codes will link to the FEIND library, and FEIND will then be responsible for containing the actual parsers and storing the nuclear data. This will remove a great deal of redundant code from applications. The task of format maintainers will also be eased, because they will be more free to release new formats. Instead of needing to create new parsers in many different activation applications, maintainers must simply ensure that a single library, FEIND, is capable of supporting the new format. Once this occurs, all of the activation codes that use FEIND will have access to the new data.

3.2 Design and Implementation

3.2.1 Overall Design and Organization

FEIND is a library, not a stand alone application. This means that it must be

linked to another application to form a complete program. During this linkage, all of FEIND's code for loading and storing data is attached to the user application, which is then capable of accessing it. Figure 3.1 below demonstrates how an activation code may interact with FEIND. Detailed descriptions of the various components of FEIND, including its interfaces, RamLib, and parsers will be introduced in subsequent sections.

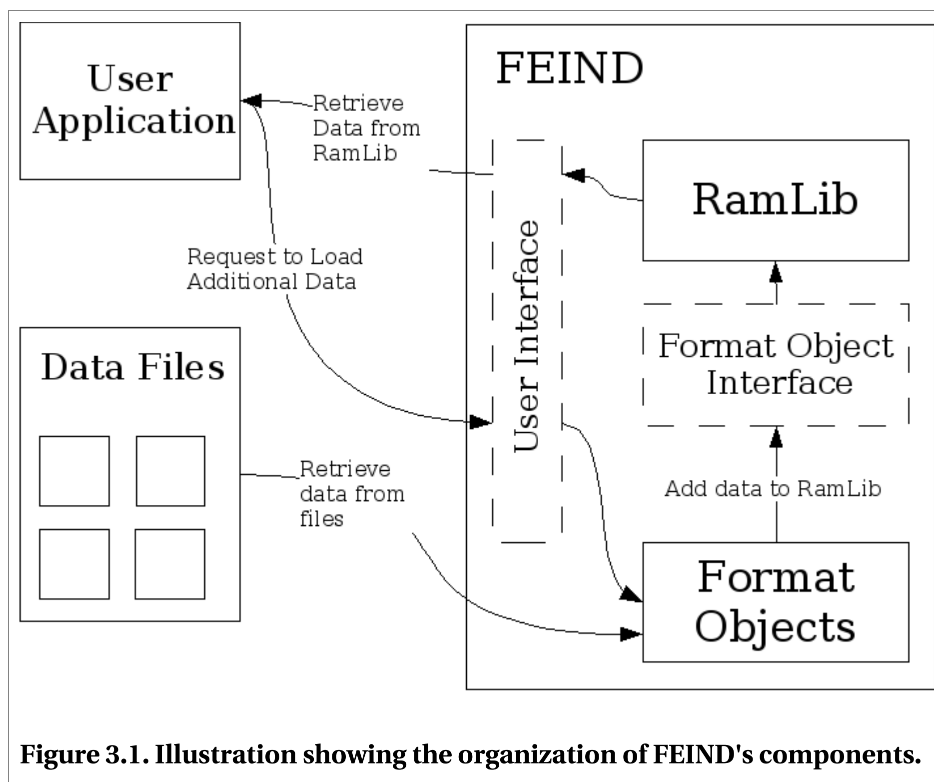

Figure 3.1. Illustration showing the organization of FEIND's components.

From a users point of view, interacting with FEIND is fairly straightforward. They begin my making a request to FEIND to load a data library into memory. The request is sent to the parser for that data format, which loads the contents of the data files. The data is then sent to the RamLib which acts as a kind of storage repository for the data. The user is then free to request the data from the RamLib, which sends it back to

the user application. From a programming standpoint, this is a very simple process to perform, and a small section of a sample application is included in appendix B to demonstrate this.

3.2.2 User Interface

Activation code developers interact with FEIND through an interface that is defined in FEIND's source code. The interface is simply a series of functions that act as the mechanism for user applications and FEIND to communicate. For example, a user may ask FEIND to load a particular data library. Or a user may request the half life of tritium from FEIND. The functions that the user application calls to make these requests *are* the interface. There are two important goals that are achieved by designing a good interface.

The first is ease of use. The interface needs to be as uncomplicated as possible. Interfaces that are difficult to use often cause frustration among users. However, there is often a trade off between ease of use and performance, and it is important to keep this in mind. Interfaces also tend to become much more complicated as the number of features in the code increases. FEIND supports many specific pieces of nuclear data that will be covered later in this report. As more different types of nuclear data are added, it can become difficult maintain the interface's simplicity. This problem can be dealt with, however, by understanding the most probable needs of the users, and by providing an appropriate default behavior.

The user interface is the *only* mechanism by which applications and FEIND communicate. This leads to the second point to be considered, which involves hiding the underlying framework from users. The design of the interface will prevent users

from interacting with FEIND's underlying data structures. The reason for this is two fold. First, it contributes to ease of use. By explicitly forbidding users to call certain functions in FEIND it helps clarify what the correct communication pathways are. Second, it prevents users from inadvertently breaking FEIND. There are certain values that are hidden behind the interface that are necessary for FEIND to operate correctly. By changing these values, users can create bugs in their own code that may be very difficult to track down.

3.2.3 RamLib

As mentioned in 3.2.2, users interact with FEIND through its interface. They use the interface to tell FEIND to perform tasks, such as loading data. However, once data is loaded into memory, it must be stored somewhere. The RamLib is a data structure that is responsible for this task. Its mission is to store data until such time as the user requests access to it. The actual data structures are hidden behind the interface, so the user has no direct access to them.

The RamLib must have the capability to store any type of data that a particular data library may contain. So far FEIND's users have only requested fairly common pieces of data, such as half-lives, cross-sections, and decay energies. However, as new types of data are needed, it may be necessary to expand the RamLib.

The RamLib acts as a central repository where all of the data is stored for users. There is only a single instance of it created when the code is run. This has greatly simplified FEIND's structure and organization, however there are a few limitations that are posed by having only one RamLib.

Most importantly, there can only be one copy of a certain piece of data loaded

at any given time. For example, suppose library A and B both contain cross sections for fissioning U^{235}. If A then B is loaded, FEIND's default behavior is to automatically load all data from both libraries. In this case, the cross-section from B will over write the cross section from A. There is no mechanism for storing both in memory. For activation codes that are, or are considering, using FEIND, this has not been a major difficulty, however it may become an issue in the future.

A second problem that arises when more and more types of data are supported. Each new addition to the RamLib will increase the memory overhead. In other words, the size of a relatively empty RamLib will grow. If the type of data is fairly common, then this is not a problem, since many libraries will support it. However, if a more obscure quantity is needed, which only exists in a single library that is relatively unused, then there may be a trade-off. Although this has not been a significant problem so far, there is potential for difficulty.

As mentioned above, in spite of these limitations the RamLib performs its task well. Both adding support for new types of data and adding functions to the interface are not difficult tasks. As FEIND, specifically the RamLib, is presently organized, it meets the requirements of all of the activation codes that currently use it. Many of the problems listed above may manifest themselves if FEIND is used to provide access to non-activation codes that still require nuclear data.

3.2.4 Parsers

The parsers in FEIND are those pieces of code that are responsible for loading individual data libraries. Data is taken from the files by the parsers, and is sent to the RamLib for storage. The parsers, like the RamLib, are stored behind FEIND's main

interface, and the user has no direct access to them.

A new parser is created for each new data format that is supported in FEIND. One of FEIND's goals, was to create a relatively straight-forward process for adding support for new data formats. As shown in figure 3.1, the parsers must interact with the RamLib. It should be noted that this introduces another interface into the code – the interface between the parsers and the RamLib. Since it is possible that other users may want to add format objects themselves, this interface must also be easy to use. Many of the same considerations introduced in 3.2.2 also apply here.

When users interact with the parsers, they usually specify the names of the files where the data is stored along with the format of the files. It is also possible for the users to give options to the parser that give them additional information regarding how to load data. For example, the CINDER [9] library contains transmutation cross-sections along with decay data. One might send the CINDER parser an option which instructs it to not load decay data. The default behavior, however, is to load all of the data into the RamLib.

3.3 Capabilities

Each data library usually contains multiple pieces of data which must be supported in FEIND. If the data already exists in FEIND, then creating the parser to support the new data becomes fairly easy. If however, new data is added that is not supported, changes need to be made to the RamLib. At present, the data consists of various transmutation cross-sections and different types of decay data.

3.3.1 Nuclear Data

Nuclear cross-sections are divided into several categories in FEIND. Currently

all of these are group-wise, not continuous. The activation programs that currently use the library are mostly interested in various transmutation cross-sections and not scattering cross-sections. Currently, the projectiles involved in these reactions are neutrons.

The cross-sections are organized in a hierarchical manner. The most detailed cross-sections are referred to in FEIND as parent/daughter/path cross-sections. This is because the reaction contains information about the parent isotope, the daughter isotope and the path. The path specifies which particles are emitted after the projectile is absorbed. In reaction 3.1 below,

$$^{7}Li\,(n;\,2n,a)\,^{2}H \tag{3.1}$$

The two emitted neutrons along with the alpha particle form the information about the path. It is possible for a reaction involving the same parent and daughter to have many different paths. Certain features, such as sequential charged particle reactions, require knowledge of the path. If this information is not needed then parent/daughter cross sections are available in FEIND.

Parent/daughter reaction cross-sections characterize the likelihood that a certain parent will yield a certain daughter regardless of the path. Some libraries contain this information directly, and in others it must be created. The FEIND parsers can sum over all of the parent/daughter/path cross-sections to yield the parent/daughter cross-sections. These are stored inside the RamLib to speed up user applications that require them. Finally, by summing over the parent/daughter cross-sections, it is possible to yield the total parent cross section, again, only if the library does not contain them.

Equation 3.2 below shows how the various cross-sections that might exist in a library can be summed together. Each cross section listed represents a parent/daughter/path cross section.

$$\begin{aligned} \sigma_1 &= \sigma\left({}^{6}Li\,(n;2n,a)\,{}^{1}H\right) \\ \sigma_2 &= \sigma\left({}^{6}Li\,(n;2n,d)\,{}^{4}He\right) \\ \sigma_3 &= \sigma\left({}^{6}Li\,(n;g)\,{}^{7}Li\right) \\ \sigma_4 &= \sigma\left({}^{6}Li\,(n;p)\,{}^{6}He\right) \\ \sigma_5 &= \sigma\left({}^{6}Li\,(n;t)\,{}^{4}He\right) \end{aligned} \tag{3.2}$$

The process for obtaining the parent/daughter, and total parent cross-section is shown below, in 3.3.

$$\begin{aligned} \sigma_{{}^{1}H} &= \sigma_1 \\ \sigma_{{}^{4}He} &= \sigma_2+\sigma_5 \\ \sigma_{{}^{7}Li} &= \sigma_3 \\ \sigma_{{}^{6}He} &= \sigma_4 \\ \sigma_{Total} &= \sum_{i=1}^{5}\sigma_i \end{aligned} \tag{3.3}$$

However, this method is only used in certain cases, usually when the total parent cross-section is not explicitly listed in the data. There are other examples of parent cross-sections as well, such as fission cross-sections. Not all parsers will fill in the total cross-section field in the RamLib, because the reactions listed in the data file may not contain the complete set needed to list the total cross-sections. Currently, all of the file formats, with the exception of the ENDF-VI-decay format contain cross-sections which must be loaded.

3.3.2 Decay Data

In addition to cross-sections, FEIND is capable of loading decay data from various libraries. The internal structure is currently designed to handle most of the

decay data stored in the ENDF-VI decay format (file 8, MT 457) [10]. This includes information about decay modes, decay constants, and decay energies. It also includes the various spectra of unstable nuclei. Of course, it is not required that individual libraries actually contain all of this data, but it is supported in the event that it exists.

3.3.3 Fission Yields

FEIND is also capable of loading fission yields. These yields can be multiplied by the total fission cross-section for an isotope to produce the corresponding parent/daughter fission cross-sections.

$$\sigma({}^{235}U\ (n, fisssion)\, {}^{135}Xe) = \sigma_{{}^{235}U\ fission} \times Yield_{{}^{135}Xe} \quad (3.4)$$

3.4 Primary vs. Secondary Reactions

One task that is not dealt with consistently among different data libraries is distinguishing between primary and secondary daughters of a reaction. This is most easily illustrated using an example such as the reaction in equation 3.5.

$$^{6}Li\,(n,p)\,^{6}He \quad (3.5)$$

The same reaction could be re-written as:

$$Li(n, {}^{6}He)\, {}^{1}H \quad (3.6)$$

which would represent the same cross-section. There may be several reasons why one would want to perform this transformation, but it essentially allows the direct calculation of the number secondary particles produced in reactions. For this reason the reaction above will be referred to as a secondary reaction.

A more precise definition of secondary vs. primary reactions is needed at this point. FEIND considers primary reactions to be those whose cross-sections sum to form the total cross-section. If FEIND added both equations 3.5 and 3.6 to form the

total cross-section, the final value would be incorrect, since the same reaction would essentially be double counted. FEIND only treats equation 3.5 as a primary reaction, and when manually determining the total cross-sections, knows to only add its contribution. Reaction 3.6 is treated as secondary, and is ignored when determining the total cross-section.

There are three different methods in which the transmutation libraries store primary, secondary and total cross-sections. The FENDL library, stored in the EAF format, does not contain any secondary reactions; all cross-sections contribute to the total. By summing over each reaction for a given parent the correct total cross-section is obtained. The IEAF [11] data library presents the opposite approach. It explicitly defines the total cross-section, and all parent/daughter cross-sections are treated as secondary. Finally, the CINDER data format defines some cross-sections to be primary and some to be secondary. The current definition of secondary as opposed to primary cross-sections used in FEIND is capable of supporting all three of these scenarios.

3.5 Data Calculations

FEIND is also designed to help users perform calculations on their data. By moving this functionality into FEIND, it is possible to reduce the amount of code that exists in user applications. The only calculation that is currently supported involves calculating reaction rate coefficients. The function LambdaEff in FEIND accepts a parent isotope and a flux. It is responsible for calculating the total reaction rate, along with reaction rates for producing each daughter of the parent.

3.6 Conclusion

The FEIND library is currently being used in the NorMC program. Its addition allowed the removal of a large volume of redundant source code. FEIND also allowed NorMC to access new data formats that were not previously supported. The AlCore program that will be discussed in the next chapter was built using FEIND from the ground up. This allowed quick and easy access to realistic nuclear data when it was required. Because of the success that these two programs have had, adding FEIND to ALARA, another activation code that is currently being used to perform real world calculations, is being considered.

FEIND is capable of supported several different data formats. The EAF 4.1 [12] format allows access to transmutation cross sections. The ENDF-VI format is used to provide decay data (although it supports many different types). CINDER data is being used to add realistic fission support. Finally, a parser for the IEAF format is currently being written.

FEIND has enabled the removal of redundant code which was one of the motivating factors. If its use is more wide spread, perhaps it can also ease the job of format maintainers as well.

4 ALCORE

4.1 Motivation and Goals

There are many codes today that do not have the capability to perform activation calculations, but where such computations might be desirable. The addition such models could allow codes to track isotopic inventories, which can allow them to perform useful calculations, such as time dependent burn up.

However, adding activation support to such codes is difficult, and to further complicate the problem, many methods currently exist for solving activation problems. These problems typically involve calculating matrix exponentials and several techniques are currently available for performing these calculations. Activation codes that currently exist usually only implement a single method, and most of these programs do not contain mechanisms for the adding additional methods.

ALCore is an activation program/library that is designed to help solve some of these problems. It is very small and contains a clean, easy-to-use interface that allows it to be linked to codes that require activation support. Furthermore, it is designed in an object-oriented way that make it easy to facilitate additional methods for solving matrix exponentials. Other programs can link to this code, and very quickly gain access to activation support.

4.2 Mathematical Models

During irradiation materials can undergo changes through transmutation reactions. The isotopes that are originally present may turn into radioactive isotopes that will subsequently decay into yet more atoms. Understanding the nature of these

transmutation and decay reactions is an important part of modeling nuclear systems. Activation problems involve solving a system of differential equations that arise when the details of these reactions are explored.

Parent isotopes can transform into other isotopes through radioactive decay. This decay is characterized by the decay constant, λ . To determine the decay constant associated with a specific parent isotope, *j*, becoming a daughter isotope, *i*, it is necessary to multiply by an associated branching ratio, *B*, which takes into account the possibility that *j* may have multiple decay modes. The reaction rate coefficient is then calculated using equation 4.1.

$$P_{j \to i} = \lambda_j B_{j \to i} \tag{4.1}$$

Isotopes can also be formed in transmutation reactions. The likelihood of an isotope producing a desired daughter is represented by the cross-section. By multiplying by the flux and integrating over all energy, it is possible to obtain a new reaction rate which is shown below for continuous and group-wise data.

$$P_{j \to i} = \int_0^{\infty} \sigma(E)_{j \to i} \phi(E) dE = \sum_{g=1}^{G} \sigma_{g,j \to i} \phi_g \tag{4.2}$$

It is then possible to combine equations 4.1 and 4.2 to create a differential equation representing the time dependent number density of the isotope of interest. This involves multiplying the reaction rates by their associated number densities and summing over all possible reactions that produce a given isotope. This is expressed in 4.3 below.

$$\dot{N}_i(t) = \sum_{j=1}^{n1} P_{j \to i}(\phi(t)) N_j(t) - \sum_{k=1}^{n2} P_{i \to k}(\phi(t)) N_i(t) \quad (4.3)$$

Here, *n1* is the number of parent isotopes and *n2* is the number of daughter isotopes. The equation above simply states that the rate of change of N_i is equal to the difference of the amount of N_i being created and the amount of N_i being destroyed. It is also possible to lump the destruction probabilities together into a new variable, *d*.

$$\dot{N}_i(t) = \sum_{j=1}^{n1} P_{j \to i}(\phi(t)) N_j(t) - d_i N_i(t) \quad (4.4)$$

It is possible to express a system of these equations, in matrix form (one for each row).Such an equation is shown below. The diagonal terms represent the destruction rates, $A_{ii} = -d_i$, and the off- diagonal terms represent the production rates, $A_{ij} = P_{ij}$.

$$\dot{\vec{N}}(t) = A\vec{N}(t)$$

The solution to this equation is the matrix exponential shown below in 4.5.

$$\vec{N}(t) = e^{At}\vec{N}(0) \quad (4.5)$$

In AlCore, a decay and transmutation tree of only a single root isotope is created. The number densities are all normalized to one, and can be multiplied by a constant to provide realistic values. Due to these two conditions, the vector containing the initial concentrations becomes:

$$\vec{N}(0) = [1 \ \ 0 \ \ 0 \ \ 0 \ldots]^T$$

AlCore uses these matrix exponentials to solve for the concentrations of individual isotopes. While these calculations represent the core of the program, they do not lend insight into how the problem is physically modeled.

Figure 4.1 shows a "tree" of isotopes that can be formed when a material is subjected to radiation. Each individual box in the tree is referred to as a "node". The example tree begins with a single node, which represents a single isotope. The isotopes contained in nodes B, E, and F are produced from A through either transmutation or decay. The isotope stored in C is produced from B and so on.

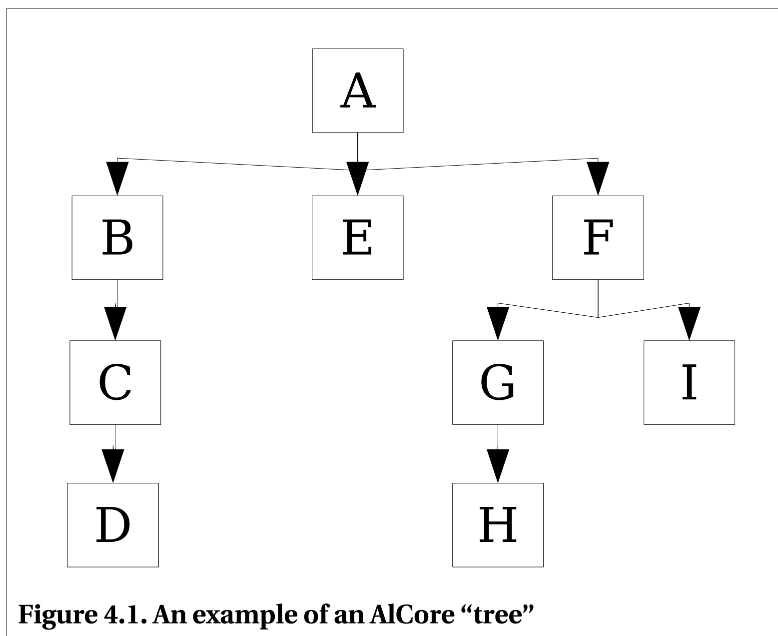

Figure 4.1. An example of an AlCore "tree"

It is important to note that each node does *not* represent a unique isotope. For example, node B may contain ^{1}H. It may undergo an (n:g) reaction and become ^{2}H which will be stored in node C. This isotope may undergo an (n:2n) reaction. In this case, node D would also contain ^{1}H.

AlCore builds a series of "chains" from this tree, and solves the matrix exponential on each to determine the isotopic content. A chain is simply a single line of connected nodes in the tree, which connects the top "root" node (A in this case) to another node. For example, nodes A, B, and C form a single chain connecting A to C.

Nodes A and B also form a unique chain. Finally a chain may also be represented by a single isotope in the case of the root node.

AlCore builds its chains in a depth first manner. Solving the matrix exponential on each chain will yield the final concentration of the node at the end of the chain. Table 4.1 below illustrates how AlCore would build these chains to solve for the concentration of each node in figure 4.1. By summing the concentrations for each node that contains a given isotope, it is possible to determine the total amount of that isotope that is produced from all pathways.

Chain Number	***Nodes Contributing to a Chain***	***Node for Which Concentration is Determined***
1	A	A
2	A, B	B
3	A, B, C	C
4	A, B, C, D	D
5	A, E	E
6	A, F	F
7	A, F, G	G
8	A, F, G, H	H
9	A, F, I	I

However, it is common for the tree be infinitely deep. To solve this problem, each time a chain is solved, the code determines not only the concentration of a given node, but the concentration of material “flowing” through a given node. Once this amount falls below some truncation limit, the code stops proceeding deeper into the tree. This truncation limit can be specified by the user.

4.2.1 Modularity

There are many different methods that can be used to solve the matrix exponential [13]. AlCore is designed to be able to support multiple models. It provides a simple interface for adding new model objects to the code. These objects simply accept some matrix, *A*, along with an associated time, and are expected to compute the exponential.

4.2.2 Taylor Series

Presently, AlCore's only model uses the Taylor Series. The exponential of a matrix, expressed by e^{At} in this case, can be re-written as shown below.

$$e^{At} = I + At + \frac{(At)^2}{2!} + \frac{(At)^3}{3!} + \frac{(At)^4}{4!} + \ldots$$

This is a simple operation, however, it can be very slow, and can take a long time to converge. This is especially true for large exponents. In an attempts to mitigate this problem, it is possible to take advantage of the fact that:

$$e^{At} = \left(e^{\frac{At}{2^n}}\right)^{2^n}$$

This allows us to reduce the size of the exponent by factors of two, then calculate the matrix exponential. Afterwards, the result is raised to the appropriate power of two. Since squaring a matrix is a relatively fast process, compared with adding terms to the Taylor series, the speed of the algorithm can be significantly increased.

4.3 Library Interface

AlCore is designed to perform the minimum requirements of an activation code. This means that it simply solves a steady-state problem on a single isotope for a

fixed amount of time. The interface is designed to be very simple and easy to use in order to appeal to the largest possible audience.

The interface requires that users provide a group-wise flux and simulation time along with a parent isotope. The group structure of the flux must match the structure of the data, of course. The user is also responsible for loading his own data into memory using FEIND (see chapter 3). As was discussed there, this is a fairly simple operation to perform as well. Finally the user must specify a truncation limit, which informs AlCore when to stop the calculation.

The calculation runs and stores the total concentration of each isotope that is produced. All of the concentrations are normalized to one, and can easily be converted to number densities. The calculation is also capable of storing the entire tree which provides information about exactly how each isotope is produced. However, for calculations with low truncation limits, the tree can become extremely large and may consume a significant amount of memory.

4.4 ALARA Comparison

AlCore was benchmarked against ALARA, another activation code that has been used for some time. The test case involved exposing iron 56 to a constant neutron flux.

The truncation limit was set to 1E-12, meaning that isotopes that were being produced in smaller quantities than this would not be reported. Also, any isotope that was reported with concentrations on or near this value would have a questionable validity, since not all pathways to a given isotope may have been included. The table shown in appendix C summarizes the results of the comparison. Only isotopes who concentrations are significantly larger than the truncation limit are shown.

4.5 Conclusion

The basic project goals were reached with AlCore. It is a very small and easy to use activation kernel that is capable of supporting multiple mathematical models. It can also be easily linked to other applications that need activation support. The results were tested against ALARA, and reasonable agreement was reached between the two codes.

However, AlCore currently only supports a Taylor series based model for calculating the solution to the matrix exponential, and has not actually been linked to other codes. An important area of future work would involve adding additional models, and finding a suitable code to link AlCore to.

5 SUMMARY

This project's main goal was to demonstrate the feasibility of improving the ease of writing activation programs by modularizing various aspects of the problem. The project focused on three major areas.

XML based input demonstrated that major sections of code could be removed by externalizing much of the work to XML parsing libraries. This helps applications use better parsers, rely on easy and standardized input file formats, and reduce the total amount of code needed to run. General guidelines were also produced for how best to write the input handler and how to maximize use of the schema. These guidelines do not require intimate knowledge of the activation program to add XML support. All of these points were demonstrated when XML support was added to NorMC using the Xerces C++ library.

FEIND similarly allowed activation applications to reduce the size of their code base. In this case routines and structures to handle nuclear data were provided by FEIND, and removed from the activation program. FEIND allowed codes that had no/limited access to nuclear data support multiple different formats. The success of this project was demonstrated by its use in NorMC and AlCore. The developers of ALARA, another activation code, are also considering using FEIND.

Finally, AlCore aimed to develop a small activation tool that could provide quick calculations. It is also written in a modular way that allows for multiple solutions of the matrix exponential to be added relatively painlessly. AlCore was tested against ALARA, and the results were fairly promising. No actual, real-world uses for this code have emerged as of yet.

The overall project was a success. Activation codes were divided into three pieces, which were each modularized. For two of the three, the solutions are already being used in other activation codes. These tools and techniques should help improve the productivity of both developers and users.

References

[1] P. Wilson, “ALARA: Analytic and Laplacian Adaptive Radioactivity Analysis,” UWFDM-1098, April 1999.

[2] "XML Introduction," http://www.w3schools.com/xml/xml_whatis.asp, 2005.

[3] “LibXML, The XML C Parser and Toolkit of Gnome,” http://xmlsoft.org/index.html, 2005.

[4] “Xerces C++ Parser,” http://xml.apache.org/xerces-c, 2005.

[5] “Extensible Markup Language (XML) 1.0 (Third Edition),” http://www.w3.org/TR/REC-xml, Febuary 2004.

[6] "C/C++ Developers: Fill Your XML Toolbox," http://www-128.ibm.com/developer/works/xml/library/x-ctlbx.html, 2001.

[7] “Document Object Model (DOM) Level 2 Core Specification,” http://www.w3.org/TR/DOM-Level-2-Core, November 2000.

[8] D. Hunter, et al, Beginning XML, Wrox Press, New York, 2003.

[9] England, T.R., “CINDER – A One-Point Depletion and Fission Product Program,” Bettis Atomic Power Laboratory report, WAPD-TM-334(Rev), 1964.

[10] V. McLane, “ENDF-102 Data Formats and Procedures for the Evaluated Nuclear Data File ENDF-6,” Brookhaven National Laboratory Review BNL-NCS-44945-01, April 2001.

[11] Y. Korovin, A. Y. Konobeev, P.E. Pereslavtsev, U.Fischer, U.V. Mollendorff, "Intermediate Energy Activation File IEAF 2001," Int. Conf. Nuclear Data for Science and Technology, Tsukuba, Japan, ND2001, October, 2001.

[12] J. Kopecky, D. Nierop, “The European Activation File EAF-4; Summary Documentation,” ECN Nuclearire Research/Faciliteiten ECN-C—95-072, 1995.

[13] C. Moler, C. Van Loan, "Nineteen Dubious Ways to Compute the Exponential of a Matrix, Twenty-Five Years Later," SIAM Review 45, 2003, 3.

Appendix A – NorMC XML Schema

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="DMCIA"
xmlns="DMCIA"
elementFormDefault="qualified">

<!-- DEFINITION OF ROOT ELEMENT -->
<!-- The root element consists of a sequence of other elements that must be defined
     in the input file, in a specific order. -->
<xs:element name="DMCIA"> <xs:complexType>
    <xs:sequence>
      <!-- Define the number of particles, nuclear data information, and total
           simulation time, only a single occurrence of this group is allowed. -->
      <xs:group ref="gen_group"/>

       <!-- Define tallies and fluxes -->
      <xs:group ref="tally_flux_group" maxOccurs="unbounded"/>

      <!-- Define control volumes mixtures -->
      <xs:group ref="cv_mixture_group" maxOccurs="unbounded"/>

      <!-- Define the sources for the problem -->
      <xs:group ref="source_group" maxOccurs="unbounded"/>
    </xs:sequence>
 </xs:complexType> </xs:element>

<!-- DEFINITION OF GROUPS USED IN THE ROOT ELEMENT -->

<!-- The “gen_group” contains several general quantities that are defined once at the
    beginning of the input file. -->
<xs:group name="gen_group"> <xs:sequence>
  <xs:element name="NPS"       type="NPSType"  /> <!-- Number of particles   -->
  <xs:element name="TotalTime" type="TimeType" /> <!-- Total simulation time -->
  <xs:element name="TransLib"  type="FileType" /> <!-- Trans. library name   -->
  <xs:element name="DecayLib"  type="FileType" /> <!-- Decay library name    -->
</xs:sequence> </xs:group>

<!-- The “tally_flux_group” contains elements that define the fluxes and tallies in
     the problem. -->
<xs:group name="tally_flux_group"> <xs:choice>
  <xs:group ref="TallyGroup" />
  <xs:element name=”FluxDef” type=”FluxDefType” />
</xs:choice> </xs:group>

<!-- This group contains variables such as control volumes and mixtures -->
<xs:group name="cv_mixture_group"> <xs:choice>
  <xs:element name="ControlVolume" type="CVType"      />
  <xs:element name="Mixture"       type="MixtureType" />
</xs:choice> </xs:group>

<!-- The “source_group” contains the sources for the problem -->
<xs:group name="source_group"> <xs:sequence>
  <xs:element name="Source" type="SourceType" />
</xs:sequence> </xs:group>
```

```
<!-- ELEMENT TYPES FOR GEN_GROUP -->

<!-- Element type for simulation time. -->
<xs:complexType name="TimeType" >
  <xs:attribute name="time" type="xs:float"/>
</xs:complexType>

<!-- Element type for defining the decay and transmutation file names -->
<xs:complexType name="FileType" >
  <xs:attribute name="path" type="xs:string" use="required"/>
</xs:complexType>

<!-- Element type for the number of particles in the problem. Although this is
     actually a long long int in the code, it must be defined as a float here, to
     allow for large numbers.-->
<xs:complexType name="NPSType">
  <xs:attribute name="num" type="xs:float" use="required"/>
</xs:complexType>

<!-- ELEMENT TYPES FOR THE TALLY_FLUX_GROUP -->

<!-- Type for the FluxDef element. This element simply consists of a sequence of
     points which define the group-wise flux. -->
<xs:complexType name=”FluxDefType”>
  <xs:sequence maxOccurs="unbounded">
    <xs:element name="Point"> <xs:complexType>
      <xs:attribute name="value"  type="xs:float" use="required" />
      <xs:attribute name="repeat" type="xs:nonNegativeInteger" use="optional"/>
    </xs:complexType> </xs:element>
  </xs:sequence>

  <xs:attribute name="name"  type="xs:string" use="required" />
  <xs:attribute name="units" type="xs:string" use="optional" />
</xs:complexType>

<!-- This group contains declarations for all of the individual tally elements that
     might exist in the input file. -->
<xs:group name="TallyGroup"> <xs:choice>
  <!-- Define the atom current tally elements -->
  <xs:element name="AtomCurrentTally" type="AtomCurrentTallyType"/>
  <xs:element name="AtomCurrentTallyEqualBinPDF" type="EqualBinPDFType"/>
  <xs:element name="AtomCurrentTallyDiscretePDF" type="DiscretePDFType"/>

  <!-- Define the atom population tally elements -->
  <xs:element name="AtomPopTallyEqualBinPDF" type="EqualBinPDFType"/>
  <xs:element name="AtomPopTallyDiscretePDF" type="DiscretePDFType"/>
</xs:choice> </xs:group>

<!-- ELEMENT TYPES FOR CV_SINK_MIXTURE_GROUP -->

<!-- The CVType defines the control volumes in the input file. The control volume can
     have exits that lead to control volumes. There can also be multiple tallies
     listed here. -->
<xs:complexType name="CVType">
  <xs:choice maxOccurs="unbounded">
    <xs:element name="CVExit" type="CVExitType"/>
    <xs:element name="Tally"/> <xs:complexType>
      <xs:attribute name="name" type="xs:string" use="required"/>
    </xs:element> </xs:complexType>
    <xs:element name="Flux"> <xs:complexType>
```

```
        <xs:attribute name="name" type="xs:string" use="required" />
    </xs:complexType> </xs:element>
  </xs:choice>

  <xs:attribute name="name" type="xs:string" use="required"/>

  <!-- Number of reactions particles are forced to have -->
  <xs:attribute name="num_forced" type="DecGE0" use="required"/>

  <!-- Amount of time particles spend in this control volume -->
  <xs:attribute name="res_time" type="DecGE0" use="required"/>
</xs:complexType>

<!-- Mixtures define different materials in the problem. They simply list different
     kza's and the relative amount of each isotope -->
<xs:complexType name="MixtureType">
  <xs:sequence>
    <xs:element name="Atom" maxOccurs="unbounded"> <xs:complexType>
      <!-- The kza of the atom in the mixture -->
      <xs:attribute name="kza"    type="xs:integer" use="required"/>

      <!-- The relative number of this atom -->
      <xs:attribute name="amount" type="DecGT0"     use="required"/>
    </xs:complexType> </xs:element>
  </xs:sequence>
  <xs:attribute name="name" type="xs:string" use="required"/>
</xs:complexType>

<!-- ELEMENT TYPES FOR SOURCE_GROUP -->

<!-- The “source_group” has only one element, which is of type SourceType. A source
     simply refers to a mixture and a control volume to start in. -->
<xs:complexType name="SourceType">
  <xs:attribute name="name"     type="xs:string" use="required"/>
  <xs:attribute name="mix"      type="xs:string" use="required"/>
  <xs:attribute name="cv"       type="xs:string" use="required"/>

  <!-- Time at which source particles should begin being generated -->
  <xs:attribute name="time"     type="DecGE0"    use="required"/>

  <!-- Relative strength of source, as compared to the other sources -->
  <xs:attribute name="strength" type="DecGT0"    use="required"/>
</xs:complexType>

<!-- DEFINE THE DIFFERENT TYPES OF TALLIES -->

<!-- This tally only requires a time, and simply tallies all atoms that cross that
     time plane -->
<xs:complexType name="AtomCurrentTallyType">
  <xs:attribute name="name" type="xs:string" use="required"/>
  <xs:attribute name="time" type="xs:string" use="required"/>
</xs:complexType>

<!-- This type represents atom current and atom population tallies that are composed
     of equally spaced time bins between low_bound and high_bound-->
<xs:complexType name="EqualBinPDFType">
  <xs:attribute name="name"       type="xs:string"             use="required" />
  <xs:attribute name="num_bins"   type="xs:nonNegativeInteger" use="required" />
  <xs:attribute name="low_bound"  type="DecGE0"                use="required" />
  <xs:attribute name="high_bound" type="DecGT0"                use="required" />
```

```
</xs:complexType>

<!-- This type is used to represent tallies which tally at discrete times -->
<xs:complexType name="DiscretePDFType">
  <xs:sequence>
    <xs:element name="PDFVal" maxOccurs="unbounded"> <xs:complexType>
      <xs:attribute name="prob" type="DecGT0" use="required"/>
    </xs:complexType> <xs:element>
  </xs:sequence>

  <xs:attribute name="name"       type="xs:string" use="required" />
  <xs:attribute name="low_bound"  type="DecGE0"    use="required" />
  <xs:attribute name="high_bound" type="DecGT0"    use="required" />
</xs:complexType>

<!-- This type is used in control volumes to specify a new control volume for
     particles to enter, and the relative probability that a particle will take that
     path. -->
<xs:complexType name="CVExitType">
  <xs:attribute name="name"        type="xs:string" use="required"/>
  <xs:attribute name="probability" type="DecGT0"    use="required"/>
</xs:complexType>

<!-- A type that restricts KZA's to valid values -->
<xs:simpleType name="kzaType">
  <xs:restriction base="xs:nonNegativeInteger">

    <!-- Smallest KZA value is H-1 -->
    <xs:minInclusive value="10010"/>

    <!-- Largest KZA value is Uun-273 -->
    <xs:maxInclusive value="1102730"/>
  </xs:restriction>
</xs:simpleType>

<!-- A type for restricting attributes to decimals greater than zero -->
<xs:simpleType name="DecGT0">
  <xs:restriction base="xs:decimal">
    <xs:minExclusive value="0"/>
  </xs:restriction>
</xs:simpleType>

<!-- A type for restricting attributes to decimals greater than or equal to zero -->
<xs:simpleType name="DecGE0">
  <xs:restriction base="xs:decimal">
    <xs:minInclusive value="0"/>
  </xs:restriction>
</xs:simpleType>
```

Appendix B – A Sample FEIND Session

The sample FEIND session below demonstrates the ease with which new data libraries can be loaded into memory. The example loads the FENDL transmutation library, then accesses the total cross-section for iron 56.

```
// Load the FEIND interface so that it can be accessed by our code:
#include "FEIND.h"

#include <vector>

int main()
{
	// lib will store information about how to access the data library.
	FEIND::LibDefine lib;

	// Add the file name to lib. This file corresponds to group-wise FENDL
	// transmutation data.
	lib.Args.push_back("~/NuclearData/fendlg-2.0_175");

	// Add the file format to the lib. In this case, the file uses the
	// EAF 4.1 format.
	lib.Format = FEIND::EAF_4_1;

	// Instruct FEIND to load the file's contents into memory:
	FEIND::LoadLibrary(lib);


	// Create a vector to store the cross-section:
	vector<double> cross_section;

	// Ask FEIND for the cross-section
	cross_section = FEIND::Library.GetPCs(26056, TOTAL_CS)


	return 0;
}
```

Appendix C – Comparison of Isotopic Concentrations Between ALARA and AlCore

Isotope	Alara Concentrations	AlCore Concentrations	Error
fe-56	9.8174E-01	9.8252E-01	.00080
fe-57	1.4781E-02	1.4833E-02	.00349
mn-55	1.3509E-03	1.3568E-03	.00443
fe-55	6.7260E-04	6.7354E-04	.00140
cr-53	4.5736E-04	4.5895E-04	.00348
fe-58	1.1860E-04	1.1936E-04	.00642
cr-54	3.0987E-05	3.1228E-05	.00780
cr-52	5.2168E-06	5.2543E-06	.00720
fe-54	1.3049E-06	1.3122E-06	.00559
mn-54	7.8274E-07	7.8876E-07	.00768
co-59	4.2962E-07	4.3362E-07	.00931
ti-50	1.3196E-07	1.3300E-07	.00787
mn-56	9.6374E-08	9.6488E-08	.00119
co-60	4.8046E-08	4.8662E-08	.01282
fe-59	2.6301E-08	2.6511E-08	.00796
v-51	1.4598E-08	1.4738E-08	.00954
ni-60	1.3194E-08	1.3389E-08	.01477
mn-53	6.0999E-09	6.1633E-09	.01039
ti-49	9.5764E-10	9.7347E-10	.01652
ni-61	2.0509E-10	2.0809E-10	.01463
fe-60	1.2835E-10	1.3761E-10	.07213