SAND REPORT

SAND 2002-2917
Unlimited Release
Printed September 2002

Developing an Event-Driven Generator for User Interfaces in the *Entero* Software

Edwin S. Wong

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of
Energy under Contract DE-AC04-94AL85000.

Approved for public release; further dissemination unlimited.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865)576-8401
Facsimile: (865)576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.doe.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800)553-6847
Facsimile: (703)605-6900
E-Mail: orders@ntis.fedworld.gov
Online order: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2002-XXXX
Unlimited Release
Printed August 2002

Developing an Event-Driven Generator for User Interfaces in the *Entero* Software

Edwin S. Wong
Sandia National Laboratories
Computational Sciences Department
P.O. Box 5800, MS 0196
Albuquerque, NM 87185-0196

Abstract

The *Entero* Software Project emphasizes flexibility, integration and scalability in modeling complex engineering systems. The GUIGenerator project supports the *Entero* environment by providing a user-friendly graphical representation of systems, mutable at runtime. The first phase requires formal language specification describing the syntax and semantics of eXtensible Markup Language (XML) elements to be utilized, depicted through an XML schema. Given a system, front end user interaction with stored system data occurs through Java Graphical User Interfaces (GUIs), where often only subsets of system data require user input. The second phase demands interpreting well-formed XML documents into predefined graphical components, including the addition of fixed components not represented in systems such as buttons. The conversion process utilizes the critical features of JDOM, a Java based XML parser, and Core Java Reflection, an advanced Java feature that generates objects at runtime using XML input data. Finally, a searching mechanism provides the capability of referencing specific system components through a combination of established search engine techniques and regular expressions, useful for altering visual properties of output. The GUIGenerator will be used to create user interfaces for the *Entero* environment's code coupling in support of the ASCI Hostile Environments Level 2 milestones in 2003.

Acknowledgments

The author thanks David R. Gardner of the Computational Sciences Department for providing feedback for this report and Joseph P. Castro and Mark A. Gonzales for their technical expertise regarding the *Entero* Software Project.

Contents

Conventions ... 6
1 Introduction ... 7
Overview of the *Entero* Software Project ... 7
Flexible Data Modeling with XML ... 7
GUIGenerator Purpose and Overview ... 7
Design Requirements ... 8
2 Overview of the *Entero* Architecture ... 8
Modules ... 9
3 Flexible Data Modeling with XML ... 9
XML Fundamentals ... 10
4 Syntax and Semantics of GUIGenerator XML ... 11
GUIGenerator Vocabulary Specification ... 11
Defining Modules ... 12
AttributeSet Element ... 12
Fixed Field Elements ... 13
Section Elements Nested within Other Sections ... 14
5 Event Driven XML Interpretation ... 14
SAX – Simple API for XML ... 14
XML Validation ... 16
DTD or XML Schema ... 16
6 Java Graphical Output ... 16
Output "Look and Feel" ... 16
Core Java Reflection ... 17
Application of Properties ... 19
Putting It All Together ... 20
7 Searching Mechanism for PrimitiveModules ... 21
Parsing Expressions ... 23
Parsing States ... 24
Constructing the Binary Tree ... 24
Complete Binary Tree Example ... 25
Evaluating the Binary Expression Tree ... 28
Rightmost Matching ... 28
8 Complete Example ... 28
9 Summary ... 33

References ... 34

Appendix A – Document Type Definition (DTD) of GUIGenerator XML ... **35**
Appendix B – XML Schema of GUIGenerator XML ... **36**
Appendix C – GUI Generating Process Diagram ... **39**
Appendix D – Sample GUI from the *Entero* Software ... **40**

Figures

1 *Entero* High Level Architectural View 8
2 *Entero* Modules 9
3 Simple XML Example 10
4 Sample XML for Records 11
5 Complete Sample of GUIGenerator XML 12
6 Fixed Field XML and GUI Output 14
7 High Level Overview of GUI Generating Process 15
8 Visual Representation of Section Properties 18
9 Conceptual Representation of Section Nesting 21
10 Example GUI Output of a PrimitiveModule 21
11 Searching Mechanism Syntax 22
12 Example Binary Tree 23
13 Boolean Expression Parsing Example, Step 1 26
14 Boolean Expression Parsing Example, Step 2 26
15 Boolean Expression Parsing Example, Step 3 27
16 List Contents of Parent Section in Complete Example 29
17 List Contents of Child Section 1 30
18 Attribute View of Geometry 0D PrimitiveModule 30
19 Component Vector of Geometry 0D PrimitiveModule 31
20 GUI Output of Child Section 1 31
21 List Contents of Child Section 2 31
22 GUI Output of Child Section 2 32
23 Final GUI Output for Complete Example 32

Conventions

`Courier 10 point font` represents a Java class
`Italicized Courier 10 point font` represents a XML element
`Bold face Courier 10 point font` represents an XML attribute

1 Introduction

Overview of the *Entero* Software Project

The *Entero* Software Project [6],[7] emphasizes flexibility, integration and scalability in aiding engineers with modeling complex systems. The modeling of these systems is done through a module-oriented, multiphysics, mixed-fidelity environment, allowing systems to be analyzed from different perspectives.

The fundamental entity of the *Entero* architecture, called the Module, emphasizes flexibility, and can be divided into two categories: `PrimitiveModule` and `CompositeModule` [3]. `Primitive-Modules` encapsulate the physical properties of a system, which are `Attributes`. For example, a thermal module encapsulates data pertaining to temperature. `CompositeModules` conceptually represent containers holding `PrimitiveModules` or other `CompositeModules`. Engineers often conceptualize systems differently from one another, and the ability to model systems as parts through Modules provides the flexibility *Entero* was designed to provide.

Flexible Data Modeling with XML

Representation of data using eXtensible Markup Language (XML) [2] complements the cross-platform flexibility of Java™. Currently, data representing *Entero* systems is stored using XML in order to prevent object-data dependencies [3]. The definition and nesting capabilities of XML inspired the idea of representing graphical user interfaces (GUIs) through XML.

GUIGenerator Purpose and Overview

Given the diverse systems that *Entero* integrates, hard coding GUIs for each system would be time inefficient and impractical, thus a dynamic interface for quickly representing a system graphically was required. The GUIGenerator software was developed to address these issues. The primary reason for the GUIGenerator is to aid developers of *Entero* by providing a general mechanism for creating a GUI for any system. In addition, engineers using the software would benefit from the ability to graphically represent a system to their specification. The dynamic GUIGenerator will become a valuable part of the *Entero* software as software requirements continue to grow.

The notion of dynamically generating GUIs from XML documents is not exclusive to *Entero*. The Sandia MAUI project [14] provides an interface for creating general Java GUIs through XML documents. Java's Forte™ software [5] represents GUIs generated from its wizards through XML. Nevertheless, MAUI, Forte and other dynamic GUI generators could not incorporate the *Entero* architecture, and so the GUIGenerator was developed.

The *Entero* GUIGenerator is a Java utility that dynamically generates a graphical user interface based on existing *Entero* system models. The GUIGenerator emulates the features of software such as MAUI, but specifically implements the *Entero* architecture and specialized GUI compo-

nents. The user specifies the appearance of the GUI through a well-formed XML document that defines visual properties and the system components to display.

Design Requirements

The GUIGenerator needed to closely resemble the architecture of *Entero*, while maintaining a conceptually simple design. The specifications called for a general method to dynamically interchange system components within a GUI through XML in order to accommodate the flexible needs of developers and engineers. GUIs needed to reflect `PrimitiveModules` and incorporate the ability to manually insert fixed components such as buttons and labels. While implementing this design goal, an additional requirement of specifying a simple but powerful XML vocabulary arose. The representative XML needed to be both simple and powerful, while conceptually paralleling the *Entero* architecture. Finally, developers needed a method to filter out irrelevant components through a searching mechanism also implemented within the XML.

2 Overview of the *Entero* Architecture

The *Entero* software environment aims to assist engineers in modeling a wide range of complex systems emphasizing flexibility, integration and scalability. The modeling of these systems is done through a module-oriented, multiphysics, mixed-fidelity environment [6].

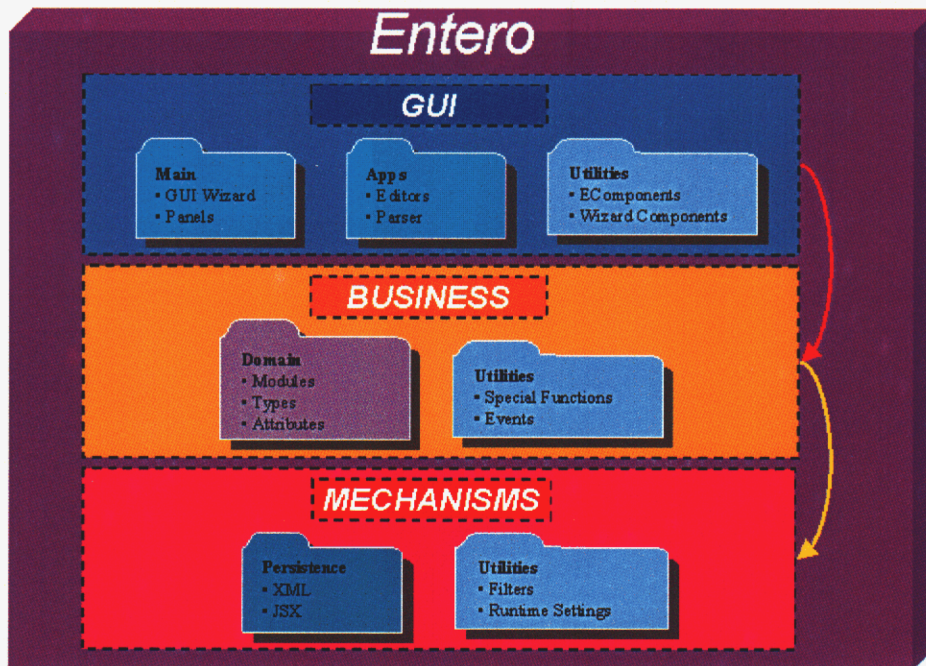


Figure 1. *Entero*'s high level architecture consists of three compositional layers.

In order to model systems of varying complexity, *Entero* needed a flexible, scalable and extensible software architecture. The *Entero* environment, from a high-level view, implements its flexible goals through its tri-layer composition, each layer dependent upon a lower layer illustrated in

Figure 1. First, the GUI layer delineates the top level of user interaction where users manipulate data, systems, mechanisms, etc. Below the GUI layer is the Business layer representing the architectural components that provide the fundamental structure to systems within *Entero*. Access is one way: the GUI layer accessing the Business layer, thereby preventing component dependencies. Finally, the Mechanism layer is the lowest layer, providing mechanisms for the Business layer. For example, the JSX mechanism [12] that is discussed in Section 5 resides within the Mechanism layer, utilized by the Business layer.

Modules

The GUI layer will be the basis of the majority of this report, but the architectural aspects of the Business layer deserve consideration. The fundamental object of the Business layer is an entity called the Module. As illustrated in Figure 2, *Entero* Modules can be divided into two categories: `PrimitiveModule` and `CompositeModule`.

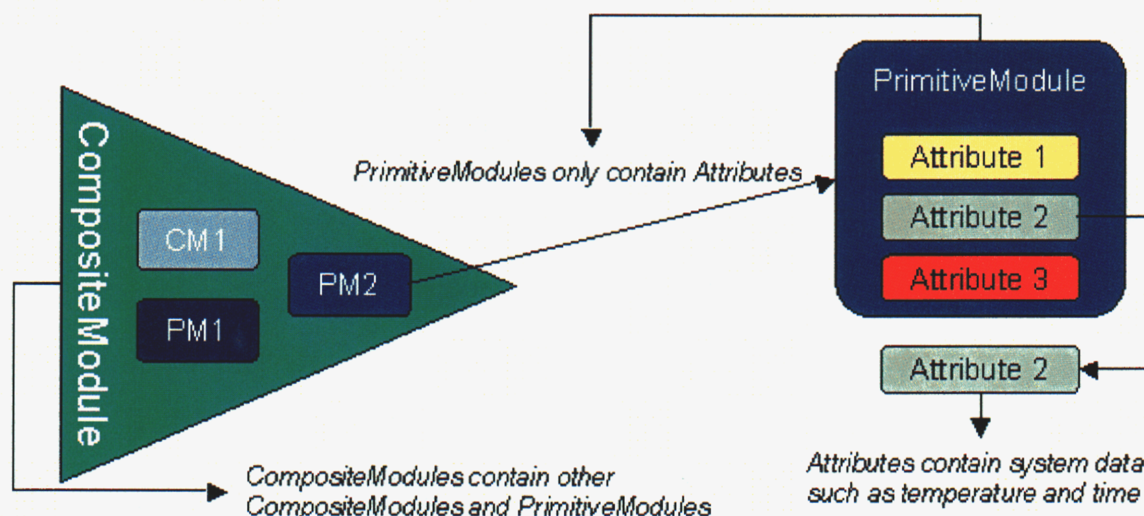


Figure 2. Illustration of the relationship between *Entero* Modules.

`PrimitiveModules` encapsulate the physical properties of a system referred to as `Attributes`. Moreover, the `PrimitiveModule` is bound to specific pre-defined categories or `ModuleType` that represent data within the module. For example, a thermal `PrimitiveModule` encapsulates data pertaining to temperature. `CompositeModules` conceptually represent containers holding `PrimitiveModules` or other `CompositeModules`, and often represent physical components. Modules provide flexibility through composition and the ability to easily add or remove parts from a particular system, one of the motivating factors for the GUIGenerator.

3 Flexible Data Modeling with XML

Representation of data using XML embodies an emerging category of computing, complementing the cross-platform flexibility of Java. With the goals of *Entero* in mind, the developers designed an improved software architecture that emulates more of a library environment than a

framework environment [7]. The *Entero* code needed to be dynamic and scalable, focusing more on flexibility than performance. Similarly, developers also saw the need to clearly divide data from code, in order to prevent object-data dependencies. All data structures are stored using XML, which provides the ability for modules to be loaded and altered at runtime.

The ability to provide engineers with the flexibility to refashion models extended down to the front end of the software. After a system has been loaded, an engineer may choose to only work with a subset of properties from the system. In addition to representing data with XML, the opportunity arose to represent GUIs with XML. In this manner, engineers would be able to configure a model without recompiling code. Moreover, XML suited the task due to its flexibility in data formatting and similarity to the *Entero* architecture. With the ability to nest and define the contents of the GUI with a powerful set of XML elements, the user easily has complete control over what is displayed.

XML Fundamentals

XML resembles the syntax of the familiar Hyper Text Markup Language (HTML) [9] native in generating Internet webpages. However, XML excels by allowing developers to define custom syntax and semantics based on XML purpose. The primary advantage of XML resides in its flexible ability to interpret any document that is well-formed. Well-formed XML consists of several features that will be referred to while describing the syntax and semantics of GUIGenerator XML. An XML tag consists of a textual name enclosed within angle brackets (< >). An XML tag must be terminated by a corresponding tag with the same name, but containing a forward slash pre-appended to the name. For example, the XML in Figure 3 depicts three tags named `GUIGenerator`, `Module` and `Section` respectively.

```
<GUIGenerator>
 <Section>
  <Module/>
 </Section>
</GUIGenerator>
```

Figure 3. Sample GUIGenerator XML demonstrating fundamentals.

Tags constitute the fundamental entities of XML, but form a more complex entity called an element. XML elements are collections of information within the scope of a beginning and terminating XML tag. For example, the sample XML in Figure 3 contains five distinct XML tags (`<GUIGenerator>`, `<Section>`, `<Module/>`, `</Section>`, `</GUIGenerator>`). However, XML elements refer to an actual tag in addition to all the information contained within the tag's scope. The `GUIGenerator` element contains three tags (`<Section>`, `<Module/>`, `</Section>`) between its beginning (`<GUIGenerator>`) and terminating (`</GUIGenerator>`) tag. The `<Module/>` tag is self terminating, indicating the beginning and termination of an element with one tag

XML elements may contain other XML elements within their scope, referred to as children, textual data, both or neither. Unless otherwise specified, XML elements may include an infinite

number of child elements, where those child elements may contain an infinite number of children, etc. In addition, XML attributes, distinct from *Entero* Attributes, provide another method of embedding information. XML attributes reside within the angle brackets of an element, contain a name followed by an equal character (=) and a value in double quotes. The example in Figure 4 illustrates a common use of XML and its nesting abilities to represent records. The `Record` element is the root of the XML document, where `Record`'s children represent aspects of the entire record. In addition, the **type** attribute provides further information, indicating that this record is for a student. The `Name` element represents a further division, where the name consists of a `First`, `Middle` and `Last` portion denoted through `Name`'s child elements.

```
<Record type="Student">
 <Name>
  <First>John</First>
  <Middle>Fred</Middle>
  <Last>Smith</Last>
 </Name>
 <Address>100 One Way</Address>
 <City>Dallas</City>
 <State>Texas</State>
 <Zip>77777</Zip>
</Record>
```

Figure 4. Simple piece of XML code to demonstrate "well-formed" syntax

4 Syntax and Semantics of GUIGenerator XML

GUIGenerator Vocabulary Specification

The XML vocabulary for describing a system in terms of graphical components needed to be simple, but powerful to prevent from overburdening users and developers. The vocabulary contains less than twenty tags, many of which are optional properties such as spacing and alignment described in Appendix A. Furthermore, the syntax needed to closely represent the final GUI product conceptually.

Each GUIGenerator XML file begins with a document root element named `GUIGenerator`, and a `Section` root element that is the only child of `GUIGenerator`. The root `Section` element provides the parent container for all other components in the GUI. The optional `Properties` element denotes a portion of the XML where the GUI appearance may be altered. Although not in the original GUIGenerator specification, user defined properties provide an enhancement toward overall flexibility. Within a particular `Section`, an infinite number of `Module`, `Field` and `Section` elements may be legally nested within a parent `Section` element in any order following the `Properties` element. For example, consider the annotated XML segment in Figure 5.

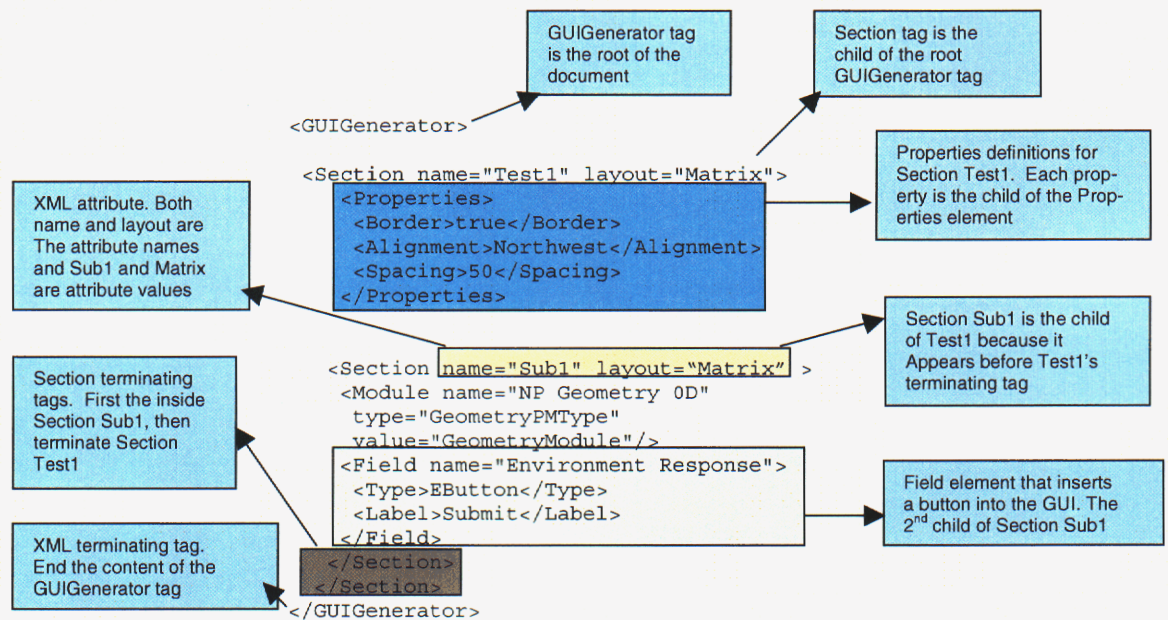


Figure 5. Sample GUIGenerator XML that illustrates the well-formed XML. Output of this XML is in Figure 9.

Defining Modules

The *`Module`* element directly relates to the `PrimitiveModule` in the *Entero* system, establishing the method of referencing system components. *`Module`* contains three XML attributes: **`name`**, **`type`** and **`value`**, all of which are required in order to pinpoint the `PrimitiveModule` in a system. The GUIGenerator `PrimitiveModules` utilize a system to reference properties encapsulated within to gain further control of internal *Entero* `Attributes`. *`Module`* elements may contain only *`AttributeSet`* elements as children, but an infinite number of children may appear. For example, the element

```
<Module name="NP Geometry 0D" type="GeometryPMType" value="GeometryModule"/>
```

represents the addition of all the `Attributes` in the `PrimitiveModule` named "NP Geometry 0D" to the output GUI.

AttributeSet Element

The *`AttributeSet`* tag furnishes the method to filter out only necessary *`Module`* properties. The *`AttributeSet`* tag contains only two attributes, both optional. First, the **`name`** attribute requires a string representing a syntactically correct boolean expression. The **`name`** attribute indicates which properties in a `PrimitiveModule` to display, using module attribute names.

Correspondingly, the **`type`** attribute of the *`AttributeSet`* element indicates which module attribute types to display. When both attributes are listed, the two attributes are applied such that a module `Attribute` must satisfy both the name and type constraints. The *`Module`* element is not

limited to one `AttributeSet` child, but can have an infinite number of `AttributeSet` elements as children. In this case, each expression in respective `AttributeSet` elements is applied such that an `Attribute` will appear if the conditions in any one `AttributeSet` element are satisfied. For example,

```
<Module name="NP Geometry 0D" type="GeometryPMType" value="GeometryModule"/>
 <AttributeSet name="Radius" type="realnumber"/>
 <AttributeSet name="Height"/>
 <AttributeSet type="url"/>
</Module>
```

indicates any `Attributes` within the NP Geometry 0D `PrimitiveModule` must satisfy at least one of the following conditions:

1. Contains the word "Radius" and is of real number type
2. Contain the word "Height" and be of any type
3. Have any name and be of URL type

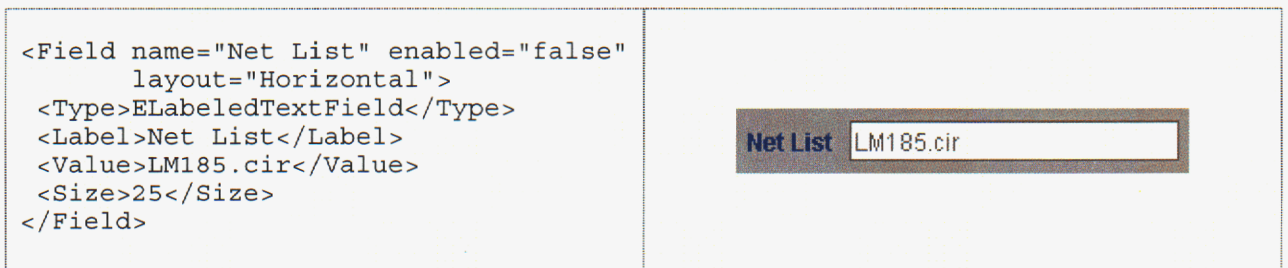
A detailed discussion on the parsing of `AttributeSet` elements occurs in Section 7.

Fixed Field Elements

During initial development, constructing the output GUI, only properties encapsulated within Modules could be displayed, without any capability for addition of other fields that could assist model representation. The idea of adding fixed components, not encapsulated in `Primitive-Modules`, adds another feature demonstrating the GUIGenerator's flexibility. Fixed element capability, resembling HTML used in generating webpages, allows insertion of GUI components not present in `PrimitiveModules`.

Similar to `Section` properties, fixed fields require user input to appear. Fixed field elements contain up to five child elements, and three XML attributes to control output appearance. First, the **`name`** attribute and `Type` element are required as the type determines the *Entero* GUI component to display and the name is used internally to reference the component for manipulation purposes such as adding `ActionListeners` to the component.

The remaining six optional properties control the appearance of the component. `Label`, `Value` and `Unit` refer to the text associated with the component, corresponding to the "look and feel" of standard components for `Attributes`. `Size` specifies the horizontal width of the component, while **`layout`** and **`enabling`** correspond to the same properties as `Section` properties. The selection of specifying the **`name`**, **`layout`** and **`enabled`** properties as attributes rather than child elements is aimed at consistency, as each of these three properties apply to other elements and are specified through attributes. Figure 6 demonstrates the addition of a field through XML and the corresponding GUI output.



```
<Field name="Net List" enabled="false"
       layout="Horizontal">
 <Type>ELabeledTextField</Type>
 <Label>Net List</Label>
 <Value>LM185.cir</Value>
 <Size>25</Size>
</Field>
```


Figure 6. Sample XML field declaration and corresponding GUI output.

Section Elements Nested within Other Sections

The ability to group objects together is critical in modeling a system, which groups components together within the context of a *`Module`*. The *`Section`* element and its corresponding Java GUI container conceptually represent a `CompositeModule`.

During the parsing of a *`Section`* element, if another *`Section`* element that is a child of the current *`Section`* element is encountered, the encountered child *`Section`* element is processed immediately before continuing with the parent element, through a recursive method call. A Java `Box` container is the output of the `processSection()` method, where the `Box` itself may be handled as a component added to another container. Therefore, the recursive call of a child *`Section`* element results in another GUI component that will be added to the collection of components for the parent section. Only when the parent section has processed all child elements, and collected all components, does the GUIGenerator construct a *`Section`*.

5 Event Driven XML Interpretation

XML is the mechanism of data persistence for *Entero* models. Within the *Entero* architecture, data stored through well-formed XML becomes transformed into Java through the JSX mechanism [12]. However, XML associated with the GUIGenerator requires parsing, but does not translate into Java objects directly.

The high level process of generating a GUI using the GUIGenerator is shown in Figure 7. The GUIGenerator implements the JDOM [13] parser, one of many implementation of the W3C XML specification [4]. The general idea of JDOM exists in providing a lightweight, conceptually simple method of representing an XML document. Furthermore, JDOM exemplifies a "Java-centric, high performance" [13] method of handling XML. The centralization of JDOM with Java greatly increases usability within Java, despite the lack of usefulness in other platforms. JDOM focuses around the Java developer, and elegantly represents an XML document through conceptually discreet classes and methods. In addition, JDOM parses and generates native Java objects such as `java.util.List` or `java.util.Map`.

SAX – Simple API for XML

JDOM combines the many parsing techniques available, however, GUIGenerator performs primarily the SAX [17] oriented features. The development of the SAX emerged due to the inefficiency of the other primary parsing method, the Document Object Model (DOM) [2]. DOM requires significant memory overhead due to the need to traverse XML documents twice. Although useful parsing using DOM exists, SAX better achieves the assignment of parsing documents of significant length.

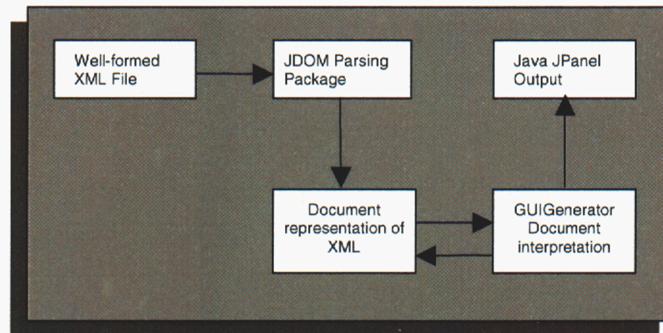


Figure 7. Overview of GUI generating process

Parsing begins though instantiating one of the `SAXBuilder` constructors. Two optional parameters may be supplied, a string indicating name of the parser class to use and a boolean flag indicating whether validation should occur. The GUIGenerator simply utilizes the default parser `org.xml.sax.XMLReader`, part of the Apache Xerces package [1]. Once instantiated, the `build()` method converts an input stream into an `org.jdom.Document`. The `Document` class defined by JDOM models the structure of an XML document by defining methods to access element names, attributes, children and element data. Reference to the XML interpretation process as "event driven" corresponds to the ability to extract XML document information when desired through JDOM methods, gathering element names, attributes, children and data rather than interpreting the entire document at once.

Once the SAXBuilder constructs the document tree from XML input, GUIGenerator extracts elements recursively beginning with the parent *`Section`* element. Utilizing the `getChildren()` method in the `Element` class returns a Java `List` storing all the child elements of the parent *`Section`* element, while preserving the order in which elements occur in the document. The algorithm for processing a *`Section`* element begins by checking for an initial *`Properties`* element, and storing the children of the *`Properties`* element in a hashtable if present. Accessing properties in the hashtable occurs in the `addComponents()` method, where the application of properties occur with the addition of components to a `Box` container. Refer to the Document Type Definition (DTD) of the XML vocabulary in Appendix A for a complete set of *`Section`* properties.

Once the collection of *`Section`* properties has been completed, the next step requires extracting all remaining child elements, including any combination of *`Module`*, *`Field`* and nested *`Section`* elements. An iterator traverses the remaining elements in the list, extracting information from respective elements through member methods. *`Section`* elements nested within other *`Section`* elements are processed recursively, in order to preserve user defined *`Section`* nesting.

XML Validation

The ability to verify the syntax of an XML file before parsing into a `Document` represents another advantageous feature of the JDOM package incorporated with SAX parsing. JDOM locates malformed XML syntax, signaling mistakes to the user before execution begins. This validation mechanism prevents costly runtime errors during XML processing, saving time with error handling.

DTD or XML Schema

Document Type Definitions (DTDs) [2] and XML Schemas [2] are both techniques for verifying XML syntax. Earlier, the GUIGenerator XML vocabulary was described using a Document Type Definition that clearly identified elements and functions. However, the validation conducted through JDOM references an XML Schema, or a model document defining the XML structure. The schema approach emerged over the DTD approach for three reasons:

- **User Defined Data Types** – Schemas provide the ability to define custom data types enhancing the flexibility of XML. For example, the alignment property permits only one of eight values (North, South, West, East...). XML schemas implement this requirement by creating an enumeration type permitting one and only one of the eight possible values.

- **Reusable Groups** – XML often presents itself through nested groups of data. Consider the technique of defining properties. Indication of properties to follow occurs through the *`Properties`* element, followed by at most nine distinct elements, children of *`Properties`*. Each possible child of *`Properties`* may be grouped, and reused through a single definition of the group.

- **Uniformity** – DTDs are written in Standard Generalized Markup Language (SGML) [2], but XML schemas are written in XML. Furthermore, validation using DTDs in JDOM requires an extra line of XML in the input file referencing the DTD.

Although the XML language defined by GUIGenerator could have easily been represented through the complete DTD given in Appendix A, but the complete XML schema, given in Appendix B, provides extra enhancements DTDs lacked.

6 Java Graphical Output

Output "Look and Feel"

There are nine tags and two attributes in the GUIGenerator vocabulary that control the appearance of the GUI output for a particular *`Section`*. First, the two attributes of the *`Section`* element, **`name`** attribute and **`layout`** attribute provide the foundation to a *`Section`*. The **`layout`** attribute, as implied, controls the layout of a given *`Section`*, but may only hold one of three values: "Vertical", "Horizontal" or "Matrix".

The Vertical layout places all GUI components in a panel linearly in a single column.

The Horizontal layout, the default layout for a `Section`, places GUI components linearly in a single row.

The Matrix layout attempts to construct a panel by arranging components in a configuration that best resembles a square. Computing the square root of the number of components, and finding the next largest whole number when the square root contains decimal yields the number of columns. Once the number of columns is determined, taking the number of components and dividing by the number of columns yields the number of rows in a matrix layout.

The **`name`** attribute of the `Section` element, although optional, controls the title of the border surrounding the section. In cases where the border exists, but no `Section` name is specified, the titled border simply becomes a standard line border. A visual representation of `Section` properties is provided in Figure 8.

The nine property tags are optional, but must appear in a uniform order, as indicated in Appendix A.

- `Border` - Takes a boolean value (true/false) and simply indicates whether a `Section` border is present.

- `Alignment` – Indicates how child components in a section are to be arranged. The alignments are implemented as nautical directions for simplicity. For example, a "North" alignment will attempt to arrange components toward the top, while a "West" alignment will attempt to arrange components to the left. Furthermore, combination directions such as "Northwest" will attempt to push components toward the top left corner.

- `HGap` – Determines the number of pixels between components arranged horizontally.

- `VGap` – Determines the number of pixels between components arranged vertically.

- `Spacing` – Determines the number of pixels to place at the perimeter of the section. When a border is present, the extra spacing is placed outside of the perimeter components, but inside of the border.

- `WestInset, EastInset, NorthInset, SouthInset` – Overrides the spacing property for a given perimeter. For example, the `WestInset` element specifies the number of pixels components must be separated from the left edge of a section.

Core Java Reflection

The representation of Java objects as well-formed XML elements relies heavily on the ability to translate text into objects at runtime, known as reflection. The Core Reflection API [11] distributed by Sun Microsystems Java 2 Platform Standard Edition [10] describes two application cate-

gories that utilize reflection. The first set includes applications requiring access to all members of a particular class utilized primarily in sophisticated applications such as interpreters or class browsers. The second set includes applications requiring access to the entire set of public members in a reflected run-time class, the category GUIGenerator employs [11].

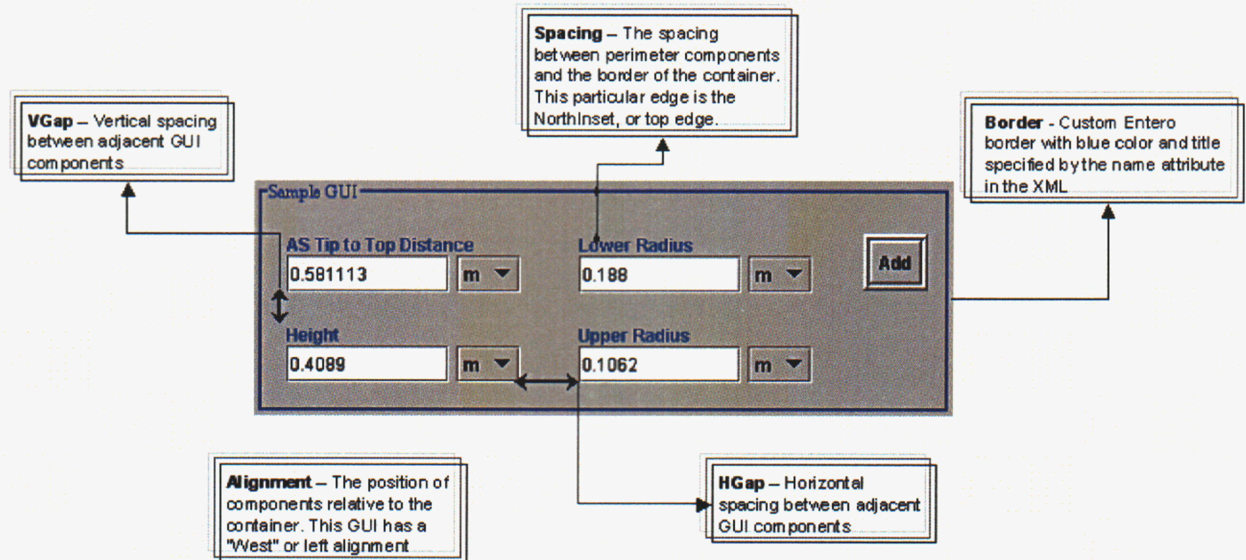


Figure 8. GUI representation of visual properties applied from XML.

The reflection algorithms required by GUIGenerator are primarily used to harvest objects from the *Entero* Business Layer, eliminating the need to interact with GUI components through the front end XML file. The XML file itself only defines the `PrimitiveModules` to display, but `Attributes` within respective `PrimitiveModules` are processed by GUIGenerator. The determination of the specific GUI component to display resides in the `AttributeType`. Currently, the *Entero* architecture supports 13 `AttributeTypes`, reflecting standard data types such as string, vector and URL. In addition, *Entero* provides the capability of encapsulating multiple values in an `Attribute`. One example of multiple encapsulations is the HASHTABLE_ENUM `AttributeType`, which maps together an integer value with a string key in order to mimic enumeration types not available in Java. *Entero* internal mechanisms utilize the integer values, but GUI components display string keys. The majority of the 13 `AttributeTypes` are associated with a corresponding `WizardComponent`. The *Entero* `WizardComponents` each implement an interface, clearly designating a uniform method of generating a Java component with `Attribute` values, while maintaining a consistent appearance. Given standard `WizardComponents`, GUIGenerator only needed to determine the `AttributeType` and determine the proper component to handle the `AttributeType`.

In order to determine the specific `PrimitiveModule` from a given system, three parameters must be predetermined from input XML: *`Module`* name, *`Module`* type and *`Module`* value. First, the specific `ModuleType` associated with the `PrimitiveModule` must be ascertained, which is used to locate `PrimitiveModules` of that type. Using the *`Module`* type as the reflected class name and *`Module`* value as the constructor parameter, invokes the `getConstructor()` and `newInstance()`

methods to return a run-time specific instance of `ModuleType`. Using this reflected instance of `ModuleType`, the `getModulesRecursively(ModuleType,level)` is invoked on the `Composite-Module` passed into the GUIGenerator. Because `CompositeModules` contain both `Primitive-Modules` and other `CompositeModules`, a `PrimitiveModule` of the reflected `ModuleType` may be nested many levels deep, requiring a recursive algorithm to find it. Once the `ModuleType` has been determined, the *`Module`* name is used to pinpoint the `PrimitiveModule` from the multitude possibly returned by the `getModulesRecursively` method, since many `PrimitiveModules` of a particular `ModuleType` may exist. Each `Attribute` in the `PrimitiveModule` is then compared with the binary expression tree, and if not filtered out, is mapped to the proper `WizardCompo-nent`.

A second use of Reflection is for generating fixed components defined by the *`Field`* element. In order to make the *`Field`* element most effective, the `getFixedField()` method uses the textual name from the *`Type`* element in the XML and attempts to reflect the corresponding component from the `entero.gui.utils` package. In addition, XML elements such as *`Label`* and *`Value`* are used as constructor parameters for creating instances of GUI components. For example, the `EButton`, a commonly used fixed field, uses the data from the *`Label`* element for a constructor parameter to generate a button displaying the corresponding text. When the element cannot be matched with a commonly used element, Reflection is used to initialize components with an empty parameter list.

Application of Properties

The initial phase of generating GUI output for a particular section begins by harvesting visual properties from the input XML document, applying the properties after all specified system components have been located. The properties, residing in a hashtable, are sent to the `addCom-ponents()` method, which constructs a `JPanel` using the hashed properties before the addition of system components. First, determination of the panel layout is followed by appropriately spacing the panel. The spacing between components along the perimeter and the edge of the panel and spacing between components is determined by accessing the spacing property in the properties hashtable, but a five pixels default is used when the spacing property is absent. In addition, the computed edge spacing is overridden when specification of a particular perimeter is present in the XML. The GUIGenerator utilizes `GridBagConstraints` [8], an object that applies output properties, to assist in controlling the `GridBag` layout manager [8], specifically manipulating the insets variable that controls spacing around individual components. The `GridBag` layout manager provides the unique ability to alter the properties of individual components in context of placement. Other layout managers in Java's Standard Development Kit [10] allow addition of components to a container controlled by preset values.

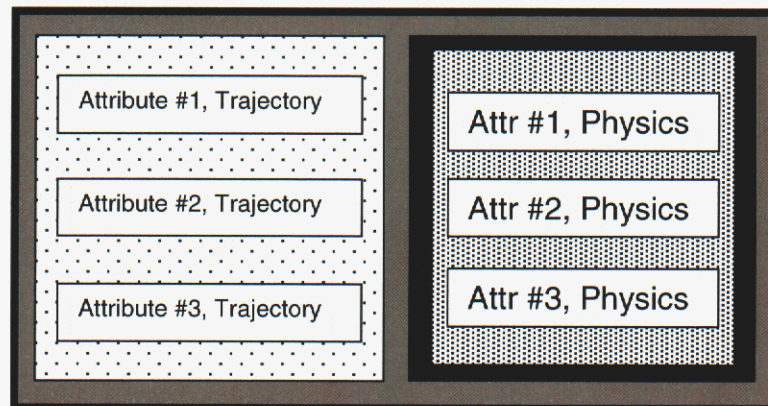
Transformed system components in the form of GUI components are iteratively placed in the output panel. The `GridBag` layout manager, required to control the position of components, utilizes a conceptual grid that requires both horizontal (x) and vertical (y) coordinates. The computation of the (x,y) position of a component requires only the number of rows. The x-coordinate is computed by dividing the element index in the linear vector by the number of rows. Similarly, the y-coordinate is computed by the element index modulo the number of rows. In addition, during each iteration, the GUIGenerator must verify whether a particular component resides on the

perimeter, where the external spacing may differ from internal vertical and horizontal gaps. The method simply checks whether the x-coordinate and y-coordinate equal zero indicating the beginning of a row or column, and the number of rows or columns equaling the maximum, indicating the end of a row or column. The panel border is the final property applied, where the standard *Entero* titled border is placed around the panel before returning to a parent *Section*.

Putting It All Together

Once location of the desired `PrimitiveModule` is completed, the `processPM (PrimitiveModule)` method determines the proper `WizardComponent` for an `Attribute`. For example, the `WizardComponent` associated with real numbers is the `WCRealNumberField`. A Java `Vector`, or resizable array of components, collects each `WizardComponent`, and is returned to the section processing algorithm where the appropriate properties are applied.

The GUIGenerator implements a combination of a `Box` and `JPanel` container to represent *Sections*, and arrange components in an aligned fashion. Neither combination alone sufficiently suited the needs of the GUIGenerator because `Box` components failed to support the flexible `GridBagLayout` and `JPanels` demonstrated poor nesting of components. `JPanels`, unlike `Boxes` contain implied insets, or invisible border spacing components that misalign *Sections* when placed together. For example, consider the GUIGenerator XML in Figure 9 demonstrating different levels of nesting.




```
<!--Define Parent Section-->
<Section layout="Horizontal">
 <!--Define Section 1 within the parent-->
 <Section name="Left" layout="Vertical">
  <Module name="Builder Trajectory" type="GeometryPMType"
          value="TrajectoryModule"/>
 </Section>
 <!--Define Section 2 within the parent-->
 <Section name="Right">
  <Section name="Right Bottom">
   <Module name="Nuclear Module - Secondary" type="PhysicsPMType"
           value="NuclearPhysicsModule"/>
  </Section>
 </Section>
</Section>
```

Figure 9. Conceptual diagram of multiple nesting with JPanels and the corresponding XML.

The example in Figure 9 will generate a GUI comprised of two sections horizontally separated by a default spacing of five pixels. The shading represents the two respective `Section`s placed in sequential order that will display the `Attributes` of two different `PrimitiveModules` in the same system. The first `Section` named "Left" and containing the Trajectory Module contains layering one level deep since no other sections are defined within the first section scope. Conversely, the second section contains layering two levels deep since one other `Section`, the "Right Bottom" `Section` exists within the "Right" `Section`. The anomaly pertaining to `JPanels` is the components in the left `Section` will be placed starting at the left corner with no extraneous spacing, but components placed in the right `Section`, nested two levels deep, will be placed within the implied insets of the inner nesting section, therefore misaligning the components. The problem becomes especially apparent when sections contain no spacing, placed directly next to one another, and during the application of borders.

The solution to the alignment problem emerges through initially adding components and applying properties within the context of a `JPanel`, but placing the finished `JPanel` within a `Box` container. The `Box` container will maintain the layout of components placed in the `JPanel`, but eliminate extraneous inset space when added by a parent container. Referring back to Figure 9, the right `Section` containing three physics `Attributes` would overlap the nested `Section` above its parent, eliminating the extraneous space in black. Although a distinct `Section` is nested within the right `Section`, the visual output is not affected with a `Box` implementation. The GUI generating process is depicted in Appendix C.

7 Searching Mechanism for PrimitiveModules

An example GUI output from the XML in Figure 5 is shown in Figure 10. The `PrimitiveModule` contains four `Attributes`: "Upper Radius", "Lower Radius", "AS Tip to Top Distance" and "Height". The four `Attributes` are of type real number and translate into specialized real number text fields.

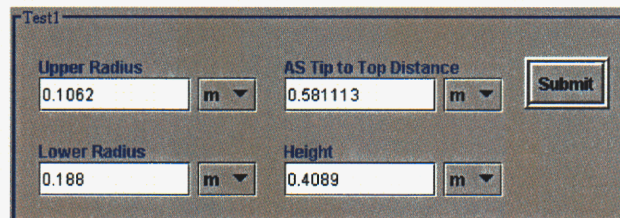


Figure 10. Example GUIGenerator output of a PrimitiveModule from XML in Figure 5.

Given the architecture of the *Entero* software, with emphasis on flexibility, the need existed to reference specific pieces of data in a system. For example, suppose we received a `CompositeModule`, obtained the indicated `PrimitiveModule` and concluded the only information that required user input were fields associated with radius. An *Entero* `PrimitiveModule` containing four `Attributes` is shown in Figure 10. However, the fact that a user only requires two of the `Attributes` to be displayed requires a method for filtering out the remaining two `Attributes`.

The `PrimitiveModule` referencing system emerged from a combination of sources. First, Internet based search engines provide the capability of so-called advanced searches. Among the features implemented within advanced searches, the ability to specify combinations with boolean capability, mandatory inclusion of keywords, mandatory exclusion of keywords and field searching, is implemented by the GUIGenerator.

Two additional requirements materialized. First, given the sophisticated framework of the *Entero* system, knowing the existence of specific attributes is difficult. Therefore, *Entero* developers needed to specify fields without knowing the complete name of an `Attribute`. Desiring a familiar mechanism, GUIGenerator allows referencing of properties based on syntactically correct regular expressions [16]. However, adding this capability required adding a mechanism to distinguish between standard keywords and regular expressions. Forward slash characters ('/'), utilized in the pattern matching language of Perl [15], are appended to the ends of regular expressions to distinguish patterns from words.

The second requirement emerged because of the characteristics of `Attribute` names. `Attributes` are often based on two or greater word permutations where white space in the context of readability and white space in the context of phrases needed to be distinguished. Therefore, the single quote character (') is a phrase delimiter in the referencing vocabulary. The GUIGenerator treats any white space inside single quotes as part of a phrase, storing the entire phrase as data in an `ExpressionNode`. Figure 11 lists the syntax for the searching mechanism.

^	Logical AND
|	Logical OR
+	Mandatory inclusion of fields with the specified keyword
-	Mandatory inclusion of specified keyword
/.../	Regular expression
`...'	Keyword Phrase
( )	Parenthesis indicate precedence of evaluation

Figure 11. Collected language syntax for generating search expressions.

Evaluating boolean expressions requires a mechanism more sophisticated than evaluating a string of tokens from left to right. The implementation of mandatory inclusion/exclusion and parenthesis alter the order of expression evaluation. Therefore, the binary tree data structure [19] was implemented in order to represent boolean expressions as a set of sub-expression. Binary trees are a collection of objects, often referred to as nodes, which contain data and pointers to two or fewer nodes in memory.

When constructing binary trees in Java, the two possible pointers are not pointers in the traditional sense, but are reference variables to other nodes in memory. One reference or child of a particular node represents a sub-expression, the root node itself represents a boolean operator and the second child node represents the second sub-expression. For example, consider the expression `(A | B) ^ (C | D)` and corresponding tree representation in Figure 12.

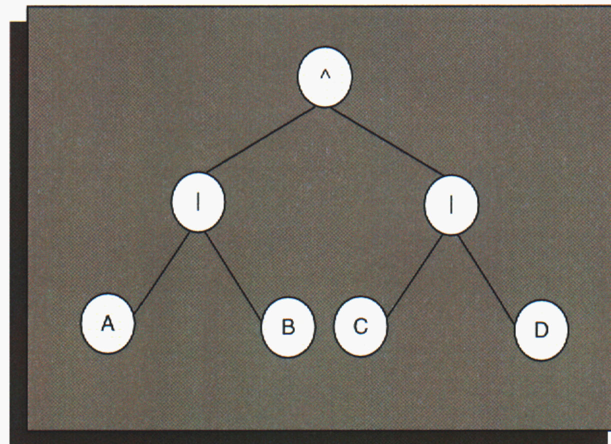


Figure 12. Conceptual view of a simple binary tree. The root node is the ^ operator, where the tree contains two sub-trees to the left and right of the root respectively.

Simply evaluating the expression from left to right fails due to the altering of evaluation order by parentheses. However, in the binary tree structure, the expression `(A | B)` is considered one sub-expression, while the expression `(C | D)` is considered another sub-expression with the ^ operator serving as the root node. This expression contains a tree that is obvious to identify because of the division into three parts. Each sub-expression also represents a tree itself. For example, the expression `A | B` contains an operator root node and two children, `A` and `B`. Similarly, the expression `C | D` can be represented in a similar manner.

Parsing Expressions

Recall from Section 4 that boolean expressions evaluated by the GUIGenerator originate from the XML attributes embedded within the `AttributeSet` elements. For each respective `AttributeSet` element, a so-called expression tree described in the preceding section represents keywords, patterns and operators. This expression parsing procedure frequently occurs in compiler theory, but GUIGenerator utilizes a simplified version because expressions are evaluated based on boolean values. Within each `AttributeSet` element, the XML attributes name and type may both appear where individual *Entero* `Attributes` require matches by both **`name`** and **`type`** in order to appear in output. Within the context of the expression tree, the expressions specified by the name and type attributes are parsed separately, and are later evaluated together through inclusive `or`.

Parsing by GUIGenerator is delegated to the `PatternMatch` class, responsible for the construction of binary trees, and the *Entero* attribute matching mechanism. Individual nodes in the binary tree are instances of the `ExpressionNode` class, which encapsulates data, reference variables to two other nodes, a boolean value indicating whether the data corresponds to an `Attribute` name or type and a second boolean value indicating enabled state of an `Attribute` should a match be found.

Expressions specified by the `name` and `type` attribute are dissected into character groups called lexemes. The `StringTokenizer` class in the `java.util` package performs this function easily, generating a buffer of lexemes that is processed iteratively. During each iteration, the parsing maintains a certain transition state [18], or the relation of the processed lexemes to the predefined syntax. For example, in the GUIGenerator syntax, the processing of a forward slash character ("/") indicates the next n characters are in the context of a regular expression, where the parser would collect all characters between slashes. This process of collecting characters within the context of a regular expression would be considered a state. The importance of determining the current state adheres to the fact that tokens in different contexts require different handling. For example, the white space character inside a single quoted phrase must be included in the data, but white space between two unquoted words is a delimiter.

Parsing States

The parsing algorithm utilized by the `PatternMatch` class is recursive. However, as mentioned in the previous section, the handling of individual lexemes is accomplished through an iterative loop. Recursion is implemented by the GUIGenerator parser through Java's first in, last out (FILO) Stack data structure.

The rules for a simple keyword or operator indicate constructing a new `ExpressionNode` with the current lexeme as data, with no left or right child, and placing the new `ExpressionNode` on the stack.

Regular expressions collect every character within a pair of forward slashes. The parsing algorithm raises the regular expression boolean flag when the first forward slash occurs, where lexemes from the `StringTokenizer` are gathered until a second forward slash is encountered. Similarly, encountering a single quote raises the phrase boolean flag, which indicates collection of lexemes until a second single quote is encountered. When the regular expression or phrase is terminated, a new `ExpressionNode` is initialized with the phrase or regular expression as data. White space not collected by a regular expression or phrase is simply ignored.

Parentheses provide the flexibility of determining evaluation order, but create an additional parsing complexity. Any expression inside a set of parentheses indicates a sub-expression that holds higher evaluation precedence. When the parsing algorithm encounters a left parenthesis, a new `ExpressionNode` with the parenthesis as data is constructed and placed on the stack, serving as a marker for the beginning of a sub-expression. Encountering a right parenthesis results in the algorithm to be discussed in the following section.

Constructing the Binary Tree

The arrangement of `ExpressionNode`'s right and left children occurs only when an expression is terminated. When the parsing algorithm reaches the end of a sub-expression (encountering a right parenthesis) or the end of the entire expression (when all lexemes have been processed), `ExpressionNodes` are popped off the stack in sets of three. The second popped node provides the root, while the first and third node serve as the right and left children of the root. The root

node is subsequently placed back on the stack, where the process is repeated until one `ExpressionNode` remains, the root node maintained by the `PatternMatch` class.

Complete Binary Tree Example

This example demonstrates the process of parsing an expression currently used by the GUIGenerator.

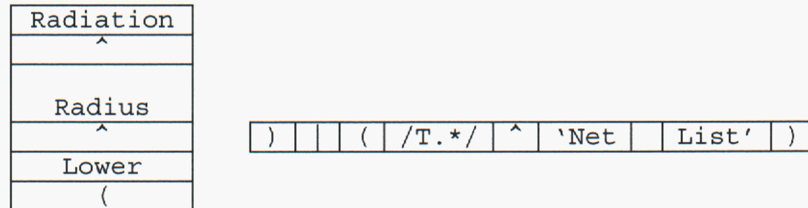
```
(Lower ^ Radius ^ Radiation) | (/T.*/ ^ 'Net List')
```

Without analyzing the syntax of the expression, a particular *Entero* Attribute must contain the word combination “lower”, “radius” and “radiation” or must contain the combination of a name starting with the letter T and containing the string “Net List”, all lexemes being case insensitive. Furthermore, the expressions inside parentheses are treated as individual expressions as will be demonstrated.

Tokenizing the expressions into lexemes will yield the following buffer:

(	Lower	^	Radius	^	Radiation	)	|	(	/T.*/	^	'Net		List'	)

White space characters, except for the space between the words “Net” and “List” are not included because they are simply ignored by the algorithm. The algorithm will place `ExpressionNodes` onto the stack unless the lexeme is a regular expression, phrase or right parenthesis. None of these conditions is satisfied until the seventh lexeme, or the right parenthesis. Therefore, the stack and lexeme buffer will contain the following elements:



The right parenthesis provides the indication that the end of the expression has been reached, and to construct a binary tree accordingly. The algorithm specifies popping nodes in sets of three, connecting the second popped node with the first popped node as the right child and the third popped as the left child as demonstrated with the binary tree in Figure 13.

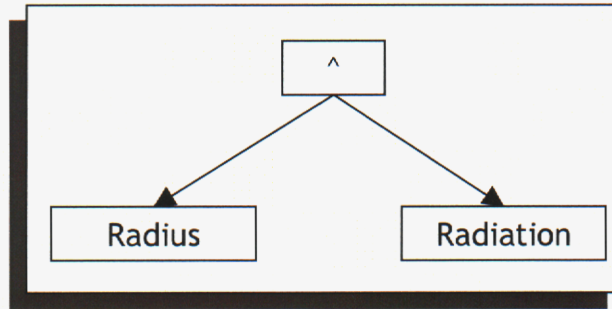
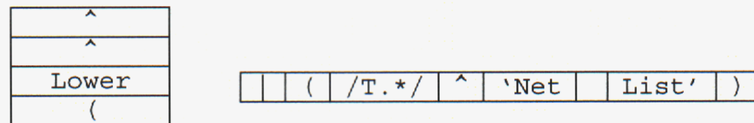


Figure 13. Binary tree generated based on the sub-expression (Radius ^ Radiation).

The root node, in this case with the data as the ^ operator, is placed back onto the stack with the following state.



The same popping mechanism is used again, yielding a new tree containing the first sub-tree and the string data "lower." The root node is again placed on the stack.

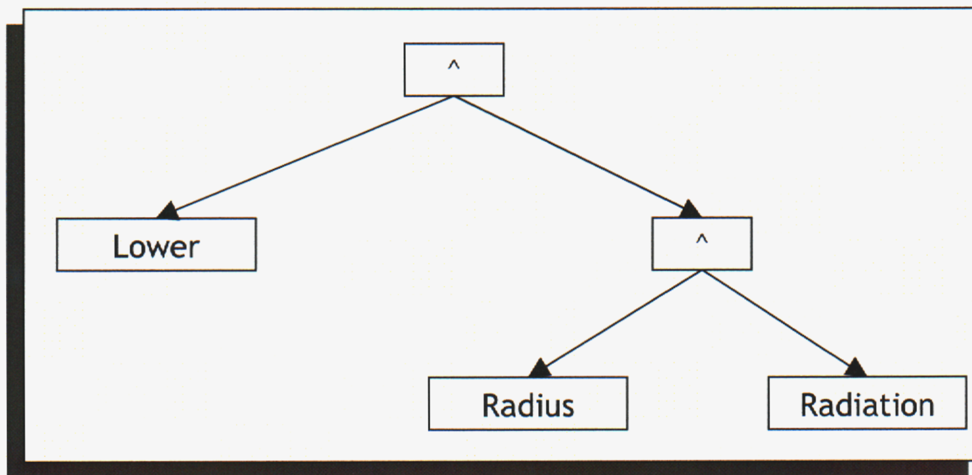


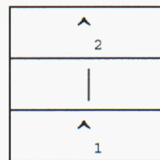
Figure 14. Binary tree generated based on the sub-expression (Lower ^ Radius ^ Radiation).

The subsequent iteration will terminate this process since the right parenthesis is located and removed off the stack. The only node that will remain is the root ^ in Figure 14.

The next irregular condition is the regular expression `/T.*/` that matches an `Attribute` beginning with the letter T. The fact that the lexeme begins with the / character automatically forces

the algorithm to check whether the last character is `/`. Because this is the case, the entire lexeme is placed on the stack and treated as a standard keyword. However, if the terminating / were absent, all lexemes would be concatenated to the regular expression until the terminating / was found. One of the proceeding lexemes will demonstrate this idea.

The lexeme 'Net indicates the beginning of a multiword phrase. Since the terminating single quote is absent, the `parseExpression()` method will continue to concatenate lexemes until the terminating single quote is located. Therefore, the phrase `'Net List'` including the white space character is placed on the stack. Finally, the terminating right parenthesis will cause tree construction similar to the first sub-expression. The stack will contain three nodes after the construction of the second sub-expression.



where $^\wedge_2$ is the root of the second sub-tree containing the phrase and regular expression, and $^\wedge_1$ is the root of the expression in Figure 14. Figure 15 demonstrates the appearance of the final binary tree.

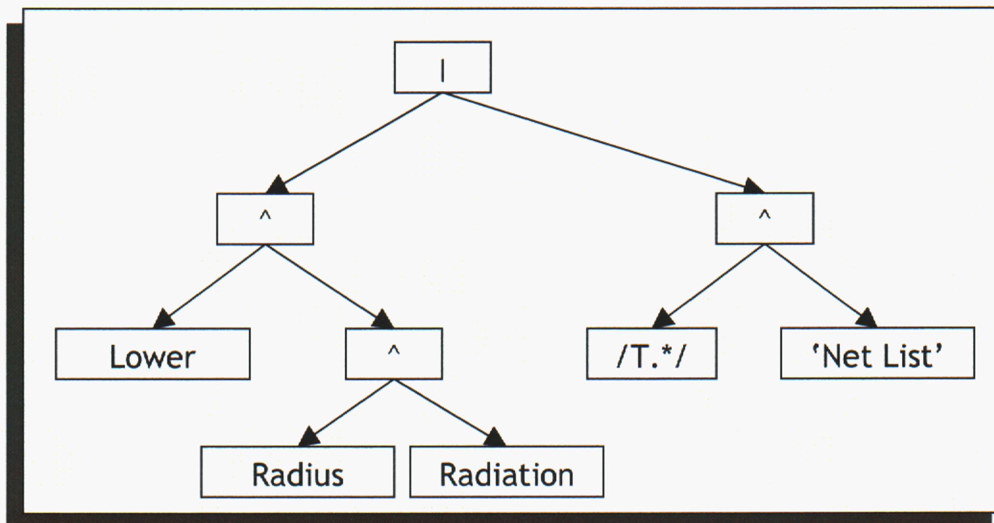


Figure 15. Full binary tree generated based on the expression (Lower ^ Radius ^ Radiation) ^ (/T.*/ ^ 'Net List').

Evaluating the Binary Expression Tree

Once the binary tree is constructed, all that is needed is a reference to the root node of the tree. With the root node containing reference variables to two other nodes, and each of these nodes contains references two nodes, evaluation of the tree is done using a recursive in-order binary tree traversal algorithm. First, the truth of each respective side of the binary tree is determined, but when either side contains null data, evaluation defaults to a true evaluation. The recursive `isMatch()` method in the `PatternMatch` class traverses until the leaves of the tree are reached where the first boolean evaluation begins. The conclusion of each evaluation results in returning a true of false response, until the truth at the root level is ascertained. The GUIGenerator then determines the presence of a particular attribute in a GUI based upon the criteria established by the `AttributeSet` tag.

Rightmost Matching

The supported searching vocabulary leaves room for ambiguity as definitions within the *`AttributeSet`* element in the XML may collide. For example, one possible collision is demonstrated by the XML below

```
<AttributeSet name="Net" enabled="false"/>
<AttributeSet name="List" enabled="true"/>
```

An `Attribute` with the name “Net List” forces the interpreter to decide which definition to enforce, since both “Net” and “List” satisfy the matching conditions.

The determination was made to enforce a rule to evaluate XML definitions with increased precedence to elements listed below others. In the preceding example, the second listed element would be enforced, enabling the GUI component related to “Net List”.

In terms of the expression tree, elements that are parsed before others are placed to the left of elements parsed later. Recalling the execution of the parsing algorithm, data is placed within `ExpressionNodes`, which are maintained on a Stack, where elements encountered first are added to the tree last. In addition, `ExpressionNodes` are removed from the Stack and placed onto the tree in sets of three nodes, using the first node as the right child, the second node as the root and the final node as the left child. In order to enforce the latest appearing rule match, the data of the corresponding rightmost `ExpressionNode` is applied.

8 Complete Example

This example demonstrates the entire process of parsing an XML file, reflecting Java objects and applying the searching mechanism in generating GUI output. Consider the following XML

```
<GUIGenerator>
 <!—Root Section element-->
 <Section name="Start" layout="Horizontal">

  <Properties>
```

```
   <Border>true</Border>
  </Properties>

  <Section name="Geometry" layout="Vertical">
   <Module name="AFF Geometry 0D" type="GeometryPMType"
           value="GeometryModule">
    <AttributeSet name="Radiation"/>
    <AttributeSet name="'Radius'" type="scalarreal" enabled="true"/>
   </Module>
   <Field name="SButton" enabled="true">
    <Type>EButton</Type>
    <Label>Submit</Label>
   </Field>
  </Section>

  <Section name="Components" layout="Vertical">
   <Properties>
    <Border>true</Border>
    <Alignment>West</Alignment>
    <HGap>10</HGap>
    <VGap>5</VGap>
    <Spacing>10</Spacing>
    <SouthInset>30</SouthInset>
   </Properties>
   <Module name="Entero 0D Lumped Thermal Code" type="ApplicationPMType"
           value="Entero Lumped Thermal Code"/>
  </Section>

 </Section>
</GUIGenerator>
```

The XML is parsed and stored into a tree data structure in the `Document` class where the root of the XML tree, the *GUIGenerator* element, can be accessed through the `getRootElement()` method. With access to the *GUIGenerator* root element, the parent *Section* element can be accessed by invoking the `getChildren()` method in the `Element` class. The recursive process of *Section*s begins with this parent *Section* element, done through the `processSection()` method. The child elements of the parent *Section* element are harvested in a list data structure. Therefore, invoking the `getChildren()` method on the parent element `<Section name="Start" layout="Horizontal">` results in a list with the following elements:

<Properties>
<Section name="Geometry" layout="Vertical">
<Section name="Components" layout="Vertical">

Figure 16. List contents after collecting the child elements of Section Start.

The parent *Section* element will contain two other *Section*s within it. Once the elements are collected in a list, an iterator checks the name of the element (*Properties*, *Module*, *Field*, *Section*). In this case, a *Properties* element and two *Section* elements are found, which require a recursive call to the `processSection()` method.

First, the children of the *Properties* element are collected and maintained in a hashtable to be used when placing components. An iterator traverses the resulting child elements and uses ele-

ment names as keys and element data as values in the hashtable. In this case, only the *Border* property is specified, which will create a border around the entire GUI with the title “Start” from the *Section* name attribute.

Next, the children of element <Section name="Geometry" layout="Vertical"> are collected in the same way as the parent *Section* element. This results in a list with the contents in Figure 17.

```
<Module name="AFF Geometry 0D" type="GeometryPMType" value="GeometryModule">
<Field name="SButton" enabled="true">
```

Figure 17. List contents after collecting the child elements of Section S1.

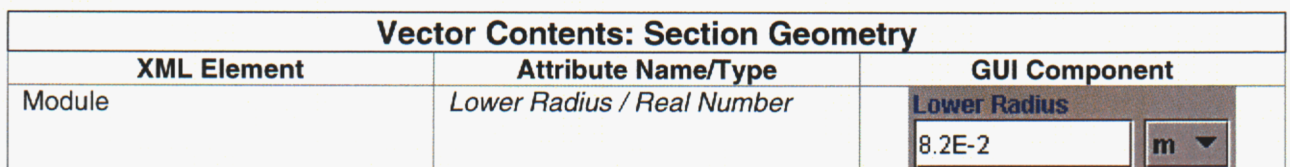
The *Module* element is reflected by the getModule() method, which returns a PrimitiveModule object that contains four Attributes.

PrimitiveModule: Geometry 0D, Type: GeometryPMType	
Attribute Name	**Attribute Type**
AS Tip to Top Distance	*Real Number*
Height	*Real Number*
Lower Radius	*Real Number*
Upper Radius	*Real Number*

Figure 18. Attribute description of PrimitiveModule Geometry 0D.

The child elements of the *Module* are collected and processed into a binary boolean expression tree. Each Attribute is then compared to this tree to determine if it appears in the GUI and if it should be enabled. In this case, the *AttributeSet* elements specify that Attributes must contain the word “Radiation” or contain the word “Radius” while being of real number type. The Geometry 0D PrimitiveModule would then have two Attributes, “Lower Radius” and “Upper Radius” matching. If an Attribute is determined to match, it is mapped to a corresponding WizardComponent, enabled according to the corresponding XML attribute and placed in a vector with other elements collected from the current call of processSection().

The second element in Figure 17 is the *Field* element, which is processed by collecting its children. The data from the *Type* element is used for reflecting the corresponding component in the entero.gui.utils package. In the case of an EButton, the data from the *Label* element is used to set the displayed text. The EButton is placed in the same vector of components as the previous Attributes from the PrimitiveModule in Figure 18. After processing the elements in Figure 17, the collection vector contains the GUI components in Figure 19.



Vector Contents: Section Geometry		
XML Element	**Attribute Name/Type**	**GUI Component**
Module	*Lower Radius / Real Number*	Lower Radius 8.2E-2 m

Module	*Upper Radius / Real Number*	Upper Radius 5.2E-2 m
Field	*N/A*	Submit

Figure 19. Vector contents after collecting the GUI components from Section S1.

The components are then added to a `JPanel`, and applied to a set of default properties since the *`Properties`* element fails to appear within the *`Section`*. The constructed `JPanel` is placed in a `Box` container to maintain alignment as described in Section 6. The GUI output is in Figure 20.

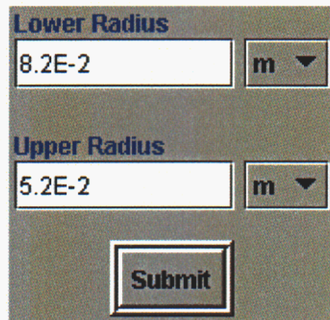


Figure 20. GUI output for Section Geometry.

The constructed GUI section is then returned to the parent section, and is handled as a component as it is placed in a vector with other generated components from elements in Figure 19.

Returning to Figure 16, the completion of processing for *`Section`* Geometry leads to the processing of the *`Section`* named "Components". The children of *`Section`* Components are harvested resulting in a list with the contents in Figure 21.

`<Properties>`
`<Module name="Entero 0D Lumped Thermal Code" type="ApplicationPMType" value="Entero Lumped Thermal Code"/>`

Figure 21. List contents after collecting the child elements of Section Components.

First, the child elements of the *`Properties`* element are processed and stored using the same process described previously. The following *`Module`* element is processed in the same way as the *`Module`* element in Figure 17. The corresponding GUI output for *`Section`* Components is in Figure 22 with the user defined properties applied.

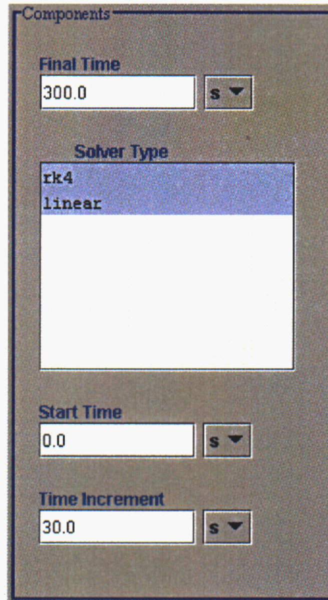


Figure 22. GUI output for Section Components.

This GUI section is also returned as a component to the parent `section`, which now contains the Box container in Figure 20 and the Box container in Figure 22. Because the processing of elements from Figure 16 has concluded, the two components associated with the parent `section` are put together to form the final GUI product in Figure 23.

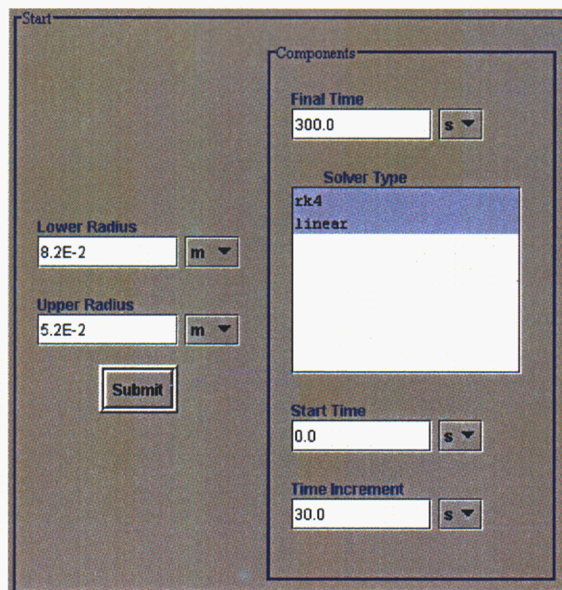


Figure 23. Final GUI output for Section.

9 Summary

The GUIGenerator Project promotes the dynamic features of the thriving technologies of XML and Java. Furthermore, it parallels the flexible architecture of the *Entero* software, making already dynamic system data mutable through the front-end graphical user interfaces. The GUIGenerator parses a well-formed XML document and accesses parsed information for generating graphical components when required.

The ideas and specification of the GUIGenerator required the ability to generate graphical output dynamically. In order to model the diverse systems integrated by the *Entero* software, the GUIGenerator needed to adapt to any current or future system in the software architecture. Furthermore, front-end GUIs are not limited to a limited set of appearances, since output GUIs are adaptable even at runtime without the overhead of recompilation. Finally, the idea of flexibility extends down to individual system properties, as users may reference Module `Attributes` through the GUIGenerator search mechanism. A GUI created by the GUIGenerator within the *Entero* software is depicted in Appendix D.

Although GUIGenerator meets its requirements, its flexibility and ease of use can be enhanced. For example, the searching mechanism can be extended to search for Modules, a feature currently being added. Other enhancements include providing some common templates for GUI appearance and providing greater user control over the appearance of sections of the generated GUI.

The GUIGenerator will be used to generate the user interfaces for the *Entero* code coupling of the NuGET neutron environment code to the Xyce™ parallel circuit code to support an ASCI Hostile Environments Level 2 milestone in 2003.

References

[1] The Apache XML Project, available at http://xml.apache.org/index.html.

[2] K. Cagle, C. Dix, D. Hunter, R. Kovack, J. Pinnock and J. Rafter. *Beginning XML, 2nd Editon.* Wrox Press Ltd., Birmingham, UK 2001.

[3] J. P. Castro, P. N. Demmie, D. R. Gardner, M. A. Gonzales, G. L. Hennigan, M. F. Young, "The *Entero* Software Architecture: Reflecting the Way Engineers Think About Systems", *Proceedings of the Summer Computer Simulation Conference 2002* (July 14-18, 2002, San Diego, CA).

[4] Extensible Markup Language, available at http://www.w3.org/XML.

[5] Forte™ for Java™ available at http://wwws.sun.com/software/sundev/jde/index.html.

[6] D. R. Gardner, J. P. Castro, P. N. Demmie, M. A. Gonzales, G. L. Hennigan, M. F. Young, "The *Entero* Project: Developing a Multifidelity System Environment for Design Engineers", *Proceedings of the Summer Computer Simulation Conference 2002* (July 14-18, 2002, San Diego, CA).

[7] D. R. Gardner, J. P. Castro, P. N. Demmie, M. A. Gonzales, G. L. Hennigan, M. F. Young, S. S. Dosanjh, "Developing a Flexible System-Modeling Environment for Engineers", *Proceedings of the Hawaii International Conference on Systems Sciences (HICSS35)* (January 7-10, 2002, Big Island, HI).

[8] C.S. Horstmann and G. Cornell. *Core Java 2, Volume 1-Fundamentals.* Sun Microsystems Press, Palo Alto, CA 1999.

[9] Hyper Text Markup Language (HTML), available at http://www.w3.org/MarkUp/.

[10] Java™ 2 Platform, Standard Edition (J2SE™), available at http://java.sun.com/j2se/.

[11] Java Core Reflection, available at http://java.sun.com/j2se/1.3/docs/guide/reflection/spec/javareflection.doc.html

[12] JSX, available at http://www.csse.monash.edu.au/bren.JSX.

[13] B. McLaughlin. *Java and XML.* O'Reilly & Associates, Inc., Sebastopol, CA 2000.

[14] MAUI, available at http://csmr.ca.sandia.gov/projects/maui/.

[15] Perl, available at http://www.perl.org/.

[16] Regular Expressions, available at http://www.opengroup.org/onlinepubs/007908799/xbd/re.html.

[17] SAX, available at http://www.saxproject.org.

[18] R. W. Sebesta. Concepts of Programming Languages Fifth Edition. Addison Wesley, Boston, MA 2002.

[19] M. A. Weiss. *Data Structures and Problem Solving Using Java.* Addison Wesley Longman, Inc., Reading, MA 1998.

Appendix A – Document Type Definition (DTD) of GUIGenerator XML

```
<!ELEMENT GUIGenerator (Section)>

<!--One section tag must be the child of the GUIGenerator tag-->
<!--Implements a pseudo-recursive approach, a section may contain a set-->
<!--of properties, an infinite number of modules and an infinite number-->
<!--of sections, nested within a parent section-->
<!ELEMENT Section (Properties, (Module | Section | Field)+)>
<!ATTLIST Section name CDATA #IMPLIED layout (Vertical | Horizontal | Matrix)
 #REQUIRED>
<!ELEMENT Properties (Border?, Alignment?, HGap?, VGap?, Spacing?, WestInset?,
 EastInset?, NorthInset?, SouthInset?)>

<!--Define the possible property tags-->
<!ELEMENT Border (#PCDATA)>
<!ELEMENT Alignment (#PCDATA)>
<!ELEMENT HGap (#PCDATA)>
<!ELEMENT VGap (#PCDATA)>
<!ELEMENT Spacing (#PCDATA)>
<!ELEMENT WestInset (#PCDATA)>
<!ELEMENT EastInset (#PCDATA)>
<!ELEMENT NorthInset (#PCDATA)>
<!ELEMENT SouthInset (#PCDATA)>

<!--Define necessary attributes of a module-->
<!ELEMENT Module (AttributeSet*)>
<!ATTLIST Module name CDATA #REQUIRED type CDATA #REQUIRED value CDATA #RE-
QUIRED>
<!ELEMENT AttributeSet EMPTY>
<!ATTLIST AttributeSet name CDATA #IMPLIED type CDATA #IMPLIED>

<!--Define Field tag-->
<!ELEMENT Field (Type?, Label?, Value?, Unit?, Size?)>
<!ATTLIST Field name CDATA #REQUIRED layout (Horizontal | Vertical) #IMPLIED
  enabled (true | false) #IMPLIED>
<!ELEMENT Type (#PCDATA)>
<!ELEMENT Label (#PCDATA)>
<!ELEMENT Value (#PCDATA)>
<!ELEMENT Unit (#PCDATA)>
<!ELEMENT Size (#PCDATA)>
```

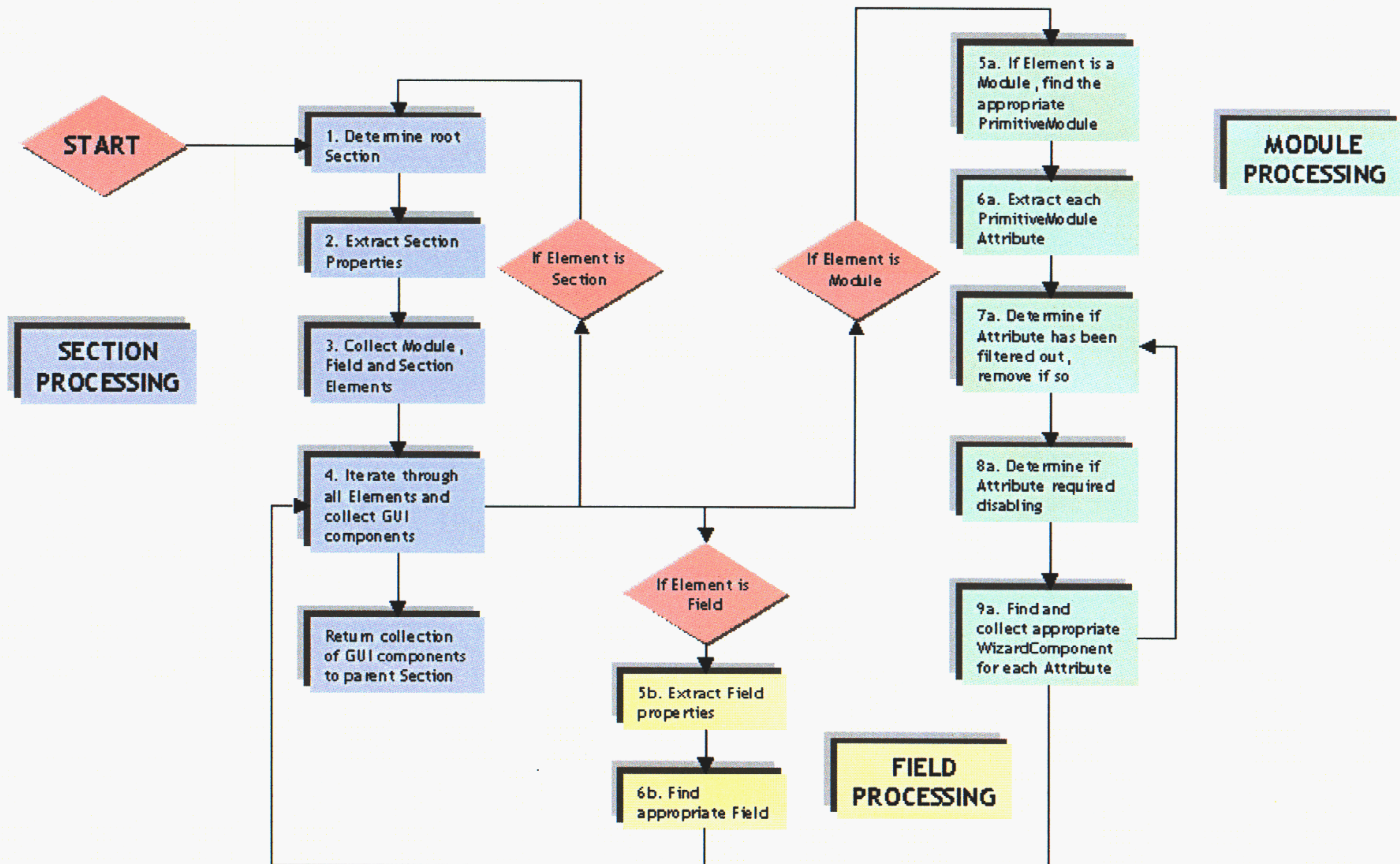
Appendix B – XML Schema for GUIGenerator XML

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

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="GUIGenerator">
  <xs:complexType>
   <xs:sequence>
    <!--Allow only one section root-->
    <xs:element ref="Section" maxOccurs="1"/>
    <!--GUIGenerator tags-->
   </xs:sequence>
  </xs:complexType>
 </xs:element>

 <!--Element 1: Section Tag-->
 <xs:element name="Section">
  <xs:complexType>
   <xs:sequence>
    <xs:element ref="Properties" minOccurs="0" maxOccurs="1"/>
    <xs:choice minOccurs="0" maxOccurs="unbounded">
     <xs:element ref="Module"/>
     <xs:element ref="Modules"/>
     <xs:element ref="Section"/>
     <xs:element ref="Field"/>
    </xs:choice>
   </xs:sequence>
   <xs:attributeGroup ref="SectionAttGroup"/>
  </xs:complexType>
 </xs:element>

 <!--Element 2: Properties Tag-->
 <xs:element name="Properties">
  <xs:complexType>
   <xs:group ref="PropertiesGroup"/>
  </xs:complexType>
 </xs:element>

 <!--Element 3: Modules inside parent-->
  <xs:element name="Module">
   <xs:complexType>
    <xs:sequence>
     <xs:element name="AttributeSet" minOccurs="0" maxOccurs="unbounded">
      <xs:complexType>
       <xs:attribute name="name" type="xs:string" use="optional"/>
       <xs:attribute name="type" type="xs:string" use="optional"/>
       <xs:attribute name="enabled" type="xs:boolean" use="optional"/>
      </xs:complexType>
     </xs:element>

    </xs:sequence>
    <!--Attributes for the module tag-->
    <xs:attribute name="name" type="xs:string" use="required"/>
    <xs:attribute name="type" type="xs:string" use="required"/>
    <xs:attribute name="value" type="xs:string" use="required"/>
   </xs:complexType>
  </xs:element>
  <!-- Element.. Module's inside parent -->
  <xs:element name="Modules">
```

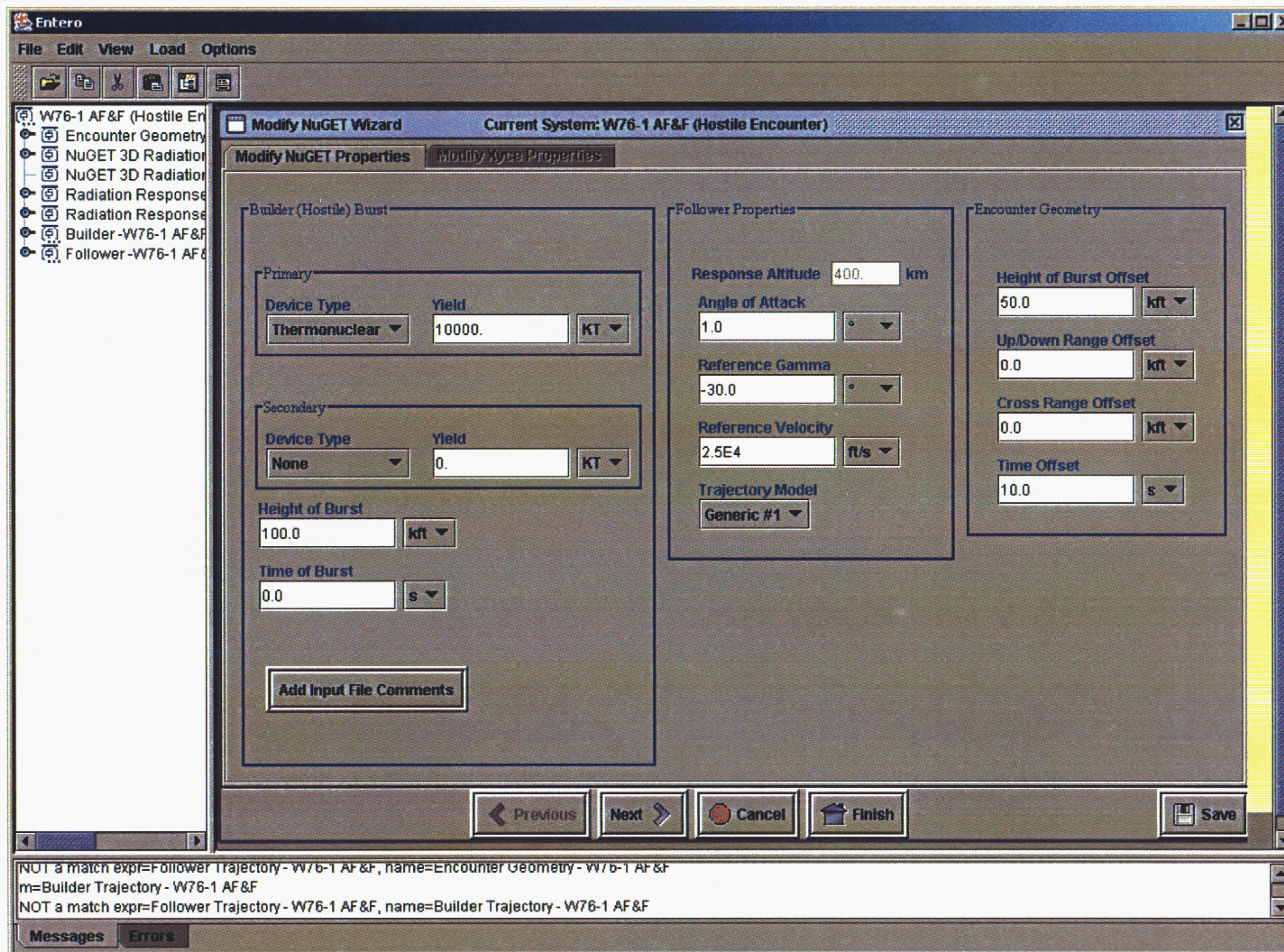
```
  <xs:complexType>
   <xs:sequence>
    <xs:element name="AttributeSet" minOccurs="0" maxOccurs="unbounded">
     <xs:complexType>
      <xs:attribute name="name" type="xs:string" use="optional"/>
      <xs:attribute name="type" type="xs:string" use="optional"/>
      <xs:attribute name="enabled" type="xs:boolean" use="optional"/>
     </xs:complexType>
    </xs:element>

   </xs:sequence>
   <!--Attributes for the Modules tag-->
   <xs:attribute name="name" type="xs:string" use="required"/>
   <xs:attribute name="type" type="xs:string" use="required"/>
   <xs:attribute name="value" type="xs:string" use="required"/>
  </xs:complexType>
 </xs:element>

 <!--Element 4: Fixed Fields-->
 <xs:element name="Field">
  <xs:complexType>
   <xs:group ref="FieldProperties"/>

   <xs:attribute name="name" type="xs:string" use="required"/>
   <xs:attribute name="layout" type="xs:string" use="optional"/>
   <xs:attribute name="enabled" type="xs:string" use="optional"/>
  </xs:complexType>
 </xs:element>

<!--Group of section attributes-->
<xs:attributeGroup name="SectionAttGroup">
 <xs:attribute name="name" type="xs:string" use="optional"/>
 <xs:attribute name="layout" default="Vertical">
  <xs:simpleType>
   <xs:restriction base="xs:string">
    <xs:enumeration value="Vertical"/>
    <xs:enumeration value="Horizontal"/>
    <xs:enumeration value="Matrix"/>
   </xs:restriction>
  </xs:simpleType>
 </xs:attribute>
</xs:attributeGroup>

<!--Group of properties-->
<xs:group name="PropertiesGroup">
 <xs:sequence>
  <xs:element name="Border" type="xs:boolean" default="false" minOccurs="0"
           maxOccurs="1"/>
  <xs:element name="Alignment" default="Center" minOccurs="0" maxOccurs="1">
   <xs:simpleType>
    <xs:restriction base="xs:string">
     <xs:enumeration value="West"/>
     <xs:enumeration value="East"/>
     <xs:enumeration value="North"/>
     <xs:enumeration value="South"/>
     <xs:enumeration value="Northwest"/>
     <xs:enumeration value="Northeast"/>
     <xs:enumeration value="Southeast"/>
     <xs:enumeration value="Southwest"/>
     <xs:enumeration value="Center"/>
    </xs:restriction>
```

```
    </xs:simpleType>
   </xs:element>
   <xs:element name="HGap" type="xs:integer" default="5" minOccurs="0"
               maxOccurs="1"/>
   <xs:element name="VGap" type="xs:integer" default="5" minOccurs="0"
               maxOccurs="1"/>
   <xs:element name="Spacing" type="xs:integer" default="10" minOccurs="0"
               maxOccurs="1"/>
   <xs:element name="WestInset" type="xs:integer" minOccurs="0"
               maxOccurs="1"/>
   <xs:element name="EastInset" type="xs:integer" minOccurs="0"
               maxOccurs="1"/>
   <xs:element name="NorthInset" type="xs:integer" minOccurs="0"
               maxOccurs="1"/>
   <xs:element name="SouthInset" type="xs:integer" minOccurs="0"
               maxOccurs="1"/>
  </xs:sequence>
 </xs:group>

 <!--Group of Fixed Field Elements-->
 <xs:group name="FieldProperties">
  <xs:sequence>
   <xs:element name="Type" type="xs:string" minOccurs="1" maxOccurs="1"/>
   <xs:element name="Label" type="xs:string" minOccurs="1" maxOccurs="1"/>
   <xs:element name="Value" type="xs:string" minOccurs="0" maxOccurs="1"/>
   <xs:element name="Unit" type="xs:string" minOccurs="0" maxOccurs="1"/>
   <xs:element name="Size" type="xs:integer" minOccurs="0" maxOccurs="1"/>
  </xs:sequence>
 </xs:group>

</xs:schema>
```

Appendix C – GUI Generating State Diagram



Appendix D – Sample GUI from *Entero* Software



Screenshot of the *Entero* software. The main panel was created by the GUIGenerator and is representative of the system displayed in the left sidebar.

DISTRIBUTION:

10 Edwin S. Wong
Texas Christian University
P.O. Box 292238
Fort Worth, TX 76129

1	MS	0321	W. J. Camp, 9200
1		0316	P. Yarrington, 9230
1		0819	E. A. Boucheron, 9231
1		0820	P. F. Chavez, 9232
1		0820	J. R. Weatherby, 9232
1		0820	P. N. Demmie, 9232
1		0316	S. S. Dosanjh, 9233
8		0316	J. P. Castro, 9233
10		0316	D. R. Gardner, 9233
1		0316	G. L. Hennigan, 9233
1		0316	S. A. Hutchinson, 9233
1		0316	J. B. Aidun, 9235
1		0739	M. F. Young, 6415
5		1137	M. A. Gonzales, 6535
1		1146	P. J. Griffin, 6423
1		1179	L. J. Lorence, 15341
1	MS	9018	Central Technical files, 8945-1
2		0899	Technical Library, 9616
1		0612	Review and Approval Desk, 9612 for DOE/OSTI