Optika: A GUI Framework for Parameterized Applications

Kurtis L. Nusbaum

Sandia National Laboratories

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

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.osti.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 ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

Optika: A GUI Framework for Parameterized Applications

Kurtis L. Nusbaum

Scalable Algorithms

Sandia National Laboratories

Mailstop 1318 Albuquerque, NM 87185-1318

klnusbaum@gmail.com

Abstract

In the field of scientific computing there are many specialized programs designed for specific applications in areas such as biology, chemistry, and physics. These applications are often very powerful and extraordinarily useful in their respective domains. However, some suffer from a common problem: a non-intuitive, poorly-designed user interface. The purpose of Optika is to address this problem and provide a simple, viable solution. Using only a list of parameters passed to it, Optika can dynamically generate a GUI. This allows the user to specify parameters' values in a fashion that is much more intuitive than the traditional "input decks" used by some parameterized scientific applications. By leveraging the power of Optika, these scientific applications will become more accessible and thus allow their designers to reach a much wider audience while requiring minimal extra development effort.

Acknowledgment

The format of this report is based on information found in [8].

Contents

Appendix

Figures

Tables

Nomenclature

Dependency A relationship between two or more parameters in which the state or value of one set of parameters depends on the state or value of another.

Dependee The parameter upon which another parameter's state or value depends.

Dependent A parameter whose state or value is determined by another parameter.

Parameter An input needed for a program.

ParameterList A class containing a list of parameters and other parameter lists.

ParameterEntry A class containing a parameter located in a ParameterList

RCP Refernce Counted Pointer [6]. RCPs refered to in this document reference the RCP class located in the Teuchos package of Trilinos.

Sublist A parameter list contained within another parameter list.

Widget A GUI element, usually used to obtain user input.

Validator An object used to ensure a particular parameter's value is valid.

Introduction

Creating a good user interface for a scientific application can be a tedious and involved task. It can also distract the application developer from the main goal of the application. Optika is an attempt to remedy this situation in a generic fashion. By drastically simplifying the process of GUI creation, Optika allows application developers to spend less time on their User Interface and more time on the core of their application. At the same time, it gives the users of these applications a much more intuitive method for supplying input via a GUI.

The purpose of this paper is to show in detail how Optika can be used to create significantly better user interfaces for scientific applications. We will include numerous examples throughout.

```
************* DIMENSION PARAMETERS ******************************************
@ -1. -1. -1. -1. 10.        Length_ref Density_ref Temp Dielec_ref VEXT_MAX
************* MESH PARAMETERS ***********************************************
@ 3         Ndim
@ 3. 2. 2.         Size_x(idim): idim=0,Ndim-1
@ 0.25 0.25 0.25         Esize_x(idim): idim=0,Ndim-1
@ -1 2         Type_bc(x0,: left, right)
(-1=IN_WALL, 0=IN_BULK, 1=PERIODIC, 2=REFLECT, 3=LAST_NODE)
@ 1 1         Type_bc(x1,: down, up)
@ 1 1         Type_bc(x2,: back, front)
```

Figure 1. A portion of the complex input file used by Tramonto

Background

A Closer Look At The Problem

We can consider a current scientific application in order to better understand the problem Optika is trying to solve. Currently, the program Tramonto has a large input file filled with complex, non-intuitive inputs. Figure 1 is just small section of this input file.

This input file is hard to read and non-intuitive, not to mention it could easily intimidate a new user. In fact, Tramonto's current input system is so complicated that it is recommended most users simply take provided sample input files and just modify them for their needs. Optika aims to solve this problem of complex input methods by allowing developers like those of Tramonto to specify their required inputs, and let Optika handle the actual retrieval of input values. The actual method of gathering input is completely abstracted away from the application developer.

Why Not Something Else?

Instead of relying on current GUI solutions to solve the above problem, Optika was created in order to address a number of constraints.

Something Simple

Current GUI frameworks like Qt (on which Optika is based), GTK+, and Cocao are incredibly robust. But because of this, they are also large and complicated. When looking for a solution to the scientific application interface problem, something simple is required. Otherwise, developers will be hesitant to use it.

Cross-Platform

Having a better user interface can lead to having a wider user base. But when adding the functionality of a GUI to an application, a developer does not want to accidentally exclude some users by using a technology that will not work on their users' platform of choice. Cocao, for example, only works on Mac OSX systems. By using the Qt Framework as its underpinnings, Optika is able to work on a wide variety of platforms.

Trilinos Integration

Optika was developed as part of the Trilinos project. Trilinos is a set of C++ libraries used extensively throughout the scientific community. Since Optika is part of Trilinos, it was desirable to have tight integration with constructs that were already in place inside of Trilinos (namely the ParameterList class). By creating our own GUI solution, Optika allows users already familiar with the Trilinos framework to take advantage of the capabilities Optika has to offer.

```
ParameterList
   ├── Parameter
   ├── Parameter
   ├── Parameter
   ├── ParameterList
   │      ├── Parameter
   │      └── Parameter
```
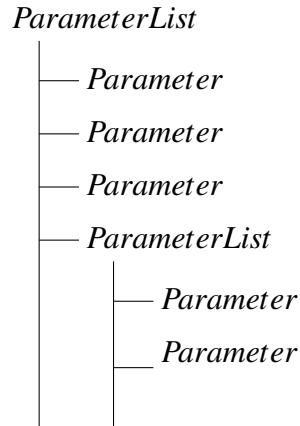
Figure 2. The hierarchical layout of an Optika GUI

Optika Overview

GUI Fundamentals

An Optika based GUI is laid out in a hierarchical fashion as shown in Figure 2. This hierarchy is made up of parameters and parameter lists. A parameter can be thought of simply as single key-value input pair for a program. A parameter list is a collection of parameters and other parameter lists. When a parameter list is part of another parameter list, it is called a sublist. Optika then uses these parameter lists to dynamically generate a GUI which is used to obtain input from the user.

Parameters in Optika are typed. Parameters can be any one of the following types:

- int and short

- float and double

- string

- boolean

- arrays of int, short, double, float, and string

- 2D arrays of int, short, double, float, and string

When editing a parameter in Optika, different "widgets" are used to change the parameters value. A widget is a graphical construct used for obtaining input from the user. For integer number types, a spin box (Figure 3(a)) is used to obtain input. For floating point number types, a line edit (Figure 3(c)) is used to obtain input. If the valid values for a string type are specified (something that will be discussed later), a combo box (Figure 3(b)) is used to obtain input for that parameter.

12

(a) A Spin Box   (b) A Combo Box   (c) A Line Edit
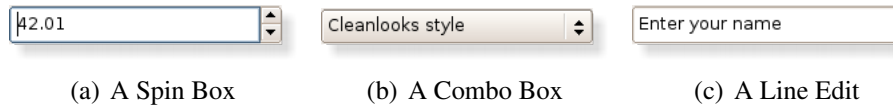
Figure 3. Some of the various widgets used for editing data in Optika [4]
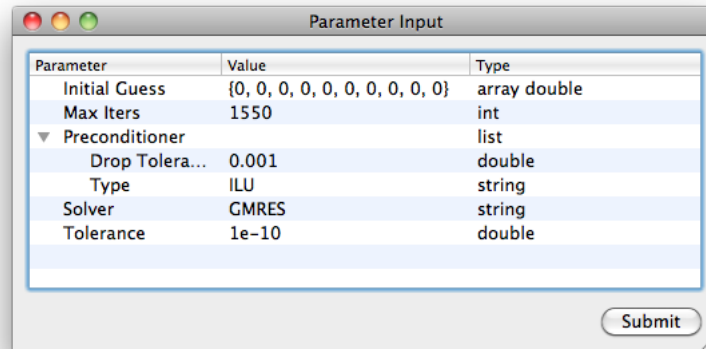
Figure 4. A basic Optika GUI

Otherwise a line edit (Figure 3(c)) is used to edit a string parameter. For booleans, a combo box (Figure 3(b)) is also used. For arrays, a pop-up box containing numerous input widgets is used. The input widgets' type is determined by the array type (e.g. for integer numerical types, a series of spinboxes is used; for string types, comboxes or lineedits are used; etc.).

When all of these components are put together, the result is a basic Optika GUI looks something like Figure 4. When the user clicks on a particular value, one of the above widgets pops up allowing them to edit the value. Once the user is finished entering input, the user clicks the action button (labeled "Submit" in Figure 4). The GUI then closes and the ParameterList used to help create the GUI now contains the input specified by the user.

Underlying Technologies

Optika is one of many packages in the Trilinos project [7] which is developed primarily at Sandia National Labs. At its core, the Trilinos project is a collection of various libraries for aiding developers of scientific applications. Trilinos is best known for its extensive collection of equation solvers, but also provides a wealth of other tools that support general high performance computing. Each package in the Trilinos project provides its own set of unique capabilities. Optika is the

Trilinos GUI package.

In order to provide GUI functionality, Optika relies on the Qt Framework [2]. Qt was chosen as Optika's backing GUI framework for several reasons:

- It is cross-platform, allowing Optika to be used in many different computing environments.

- It is mature and has a comprehensive set of development tools.

- It has a rich feature-set, permitting Optika to grow in its capabilities if necessary.

- It has been used by Sandia in the past.

- The Optika lead developer was familiar with it.

In order to configure and build Optika, the CMake [1] build system is used. This is the build system used by Trilinos as a whole, so it makes perfect sense for Optika to use it as well. In addition, CMake has support for the necessary special building tools required by Qt.

Basic Optika Usage

At the core of Optika is the ParameterList class from the Teuchos package [5], another part of the Trilinos project. ParameterLists can be constructed either from within C++ source code or using XML. In this paper, we will first start off with a basic example using C++ source code. However, once we start talking about more complex features, we will switch over to XML.

Utilizing C++ to Create A GUI

Looking at the sample Tramonto input file above, we can see how Optika could be used to create a better user interface. While examining the example input file section, we see it deals with "Dimension Control Parameters." These are a group of parameters used to define what kind of dimensions are going to be used in the rest of the input file. Figure 5 shows how we would create a GUI for these inputs.

We start by including the Optika_GUI.hpp file. This include file allows us to use most of Optika's basic functionality. Once in the main function we import several classes and functions into the namespace. We then create two ParameterList RCPs [6]. Using the sublist function, the second ParameterList is defined as a sublist of the first. We provide the lists with the names "Tramonto" and "Dimension Control Parameters." The next several lines are calls to the "set" function. Each call adds a Parameter to the ParameterList along with assigning it a default value and a documentation string (the documentation string is optional but highly encouraged). We then make the all important call to the "getInput" function. This takes the list of parameters we have created and dynamically generates a GUI which is then used to obtain input from the user. The end result is Figure 6.

Utilizing XML to Create a GUI

Using XML to declare input parameters has a number of advantages over declaring inputs in C++ source code.

- XML is a lot cleaner than the corresponding C++. The source code approach can get unruly, especially when some of the more advanced features of Optika are used.

- As an extension of being cleaner, using XML makes maintaining a user interface much simpler.

- When using XML, there is no need to recompile the entire program every time a small change is made to the GUI.

If we redo the example we made in the previous section, but using XML, we end up with the C++ code and XML code found in Figure 7 and Figure 8 respectively. The XML code is rather

```cpp
/*
// @HEADER
// ***********************************************************************
//
//         Optika: A Tool For Developing Parameter Obtaining GUIs
//                  Copyright (2009) Sandia Corporation
//
// Under terms of Contract DE-AC04-94AL85000, with Sandia Corporation, the
// U.S. Government retains certain rights in this software.
//
// Redistribution and use in source and binary forms, with or without
// modification, are permitted provided that the following conditions are
// met:
//
// 1. Redistributions of source code must retain the above copyright
// notice, this list of conditions and the following disclaimer.
//
// 2. Redistributions in binary form must reproduce the above copyright
// notice, this list of conditions and the following disclaimer in the
// documentation and/or other materials provided with the distribution.
//
// 3. Neither the name of the Corporation nor the names of the
// contributors may be used to endorse or promote products derived from
// this software without specific prior written permission.
//
// THIS SOFTWARE IS PROVIDED BY SANDIA CORPORATION "AS IS" AND ANY
// EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
// IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
// PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL SANDIA CORPORATION OR THE
// CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
// EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
// PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
// PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
// LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
// NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
// SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
//
// Questions? Contact Kurtis Nusbaum (klnusbaum@gmail.com)
//
// ***********************************************************************
// @HEADER
*/

#include "Optika_GUI.hpp"

int main(int argc, char* argv[]){
  using Teuchos::ParameterList;
  using Teuchos::RCP;
  using Teuchos::rcp;
  using Teuchos::sublist;
  RCP<ParameterList> tramontoList =
    rcp(new ParameterList("Tramonto"));
  RCP<ParameterList> dimList =
    sublist(tramontoList, "Dimension Control Parameters");

  dimList->set( "Length_ref", -1.0,
    "Indicates how length parameters will be entered");

  dimList->set( "Density_ref", -1.0,
    "Indicates how density parameters will be entered");
```
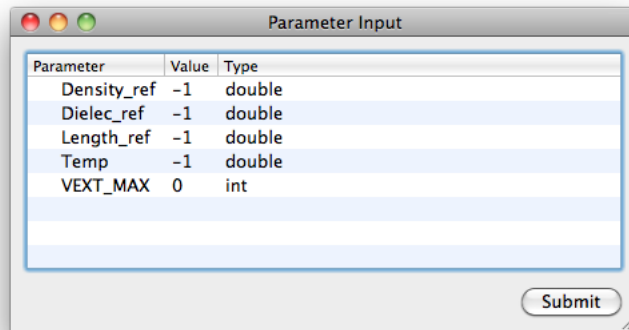
Figure 6. The final product of the code in Figure 5

straight-forward. As in the previous C++ example, we specify the name of the parameter, its default value, and a documentation string (the documentation XML attribute is optional but highly recommended, just like when using C++). When using XML, we need to explicitly specify the type of the parameter since this cannot always be inferred from XML. Creating a ParameterList hierarchy is also arguable easier in XML. ParameterLists just simply need to be nested in one another.

Once the XML file describing the input parameters has been created, the GUI is created by using a slightly different call to the getInput function. In this case, the getInput function is passed the name of the XML file and an RCP pointing to the ParameterList into which user input should be stored.

```
/*
// @HEADER
// ************************************************************************
//
//          Optika: A Tool For Developing Parameter Obtaining GUIs
//                  Copyright (2009) Sandia Corporation
//
// Under terms of Contract DE-AC04-94AL85000, with Sandia Corporation, the
// U.S. Government retains certain rights in this software.
//
// Redistribution and use in source and binary forms, with or without
// modification, are permitted provided that the following conditions are
// met:
//
// 1. Redistributions of source code must retain the above copyright
// notice, this list of conditions and the following disclaimer.
//
// 2. Redistributions in binary form must reproduce the above copyright
// notice, this list of conditions and the following disclaimer in the
// documentation and/or other materials provided with the distribution.
//
// 3. Neither the name of the Corporation nor the names of the
// contributors may be used to endorse or promote products derived from
// this software without specific prior written permission.
//
// THIS SOFTWARE IS PROVIDED BY SANDIA CORPORATION "AS IS" AND ANY
// EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
// IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
// PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL SANDIA CORPORATION OR THE
// CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
// EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
// PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
// PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
// LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
// NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
// SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
//
// Questions? Contact Kurtis Nusbaum (klnusbaum@gmail.com)
//
// ************************************************************************
// @HEADER
*/

#include "Optika_GUI.hpp"

int main(int argc, char* argv[]){
  using Teuchos::RCP;
  using Teuchos::ParameterList;
  using Teuchos::rcp;
  RCP<ParameterList> userInput = rcp(new ParameterList);
  Optika::getInput("inputs.xml", userInput);
  return 0;
}
```

Figure 7. The supporting C++ source code needed to utilize the
XML in Figure 8

```
<ParameterList name="Tramonto">
  <ParameterList name="Dimension Control Parameters">
    <Parameter name="Length_ref" value="-1.0" type="double"
      docString="Indicates how length parameters will be entered"/>
    <Parameter name="Density_ref" value="-1.0" type="double"
      docString="Indicates how density parameters will be entered"/>
    <Parameter name="Temp" value="-1.0" type="double"
      docString="Indicates how density paraemters will be entered"/>
    <Parameter name="Dielec_ref" value="-1.0" type="double"
      docString="The dielectric constant"/>
    <Parameter name="VEXT_MAX" value="0" type="double"
      docString="Max external field where DFT equations will be solved"/>
  </ParameterList>
</ParameterList>
```

Figure 8. The XML that can be used to generate the same GUI as in Figure 5

Validators

Often input parameters only have a certain set of valid values. Through the use of validators Optika allows application developers to express these parameter constraints. When a validator is placed on a parameter, the generated GUI will not accept invalid parameter values. Like parameter lists, validators were also already part of the Teuchos package before Optika was built. However, the existing set of validators was sparse. Optika took the validator constructs that were already in place and built on them, resulting in the construction of several new types of validators detailed below. These new validators were then placed in the Teuchos package.

Validators are declared in a special section of the XML file. A special ¡Validators¿ tag is declared as a direct child of the root ¡ParameterList¿ tag. In this ¡Validators¿ tag all validators are declared. Each validator must specify at least its type and an Id using the ¡Validator¿ tag. This Id can then be used *anywhere* in the rest of the XML file. To apply the validator to a parameter, simply add the validatorId XML attribute to the ¡Parameter¿ tag. The same validator can be used on multiple parameters.

Enhanced Number Validators

Perhaps one of the most important validators is the Enhanced Number Validator. It allows the application developer to specify minimum and maximum values for a numerical parameter (both inclusive). It also allows the developer to specify the "step" of an integer numerical parameter. The step of a numerical parameter is the amount by which the parameter's value will be increased or decreased when the user clicks the up and down arrows on the spinbox being used to edit the parameter's value. For non-integer numerical parameters, the Enhanced Number Validator also allows the developer to declare with what precision the parameter's value should be displayed to the user.

The XML in Figure 9 shows an example of using an Enhanced Number Validator to validate a parameter of type double. In the example, we set the minimum acceptable value to be 0 and the maximum to be 10. We also set the precision to two decimal places. The validator is given the Id of 1 and applied thusly to the "Double Parameter" parameter using the validatorId XML attribute. Notice how the "type" XML attribute of the validator is set to "EnhancedNumberValidator(double)". If we were validating a parameter of type int, this XML attribute would be set to "EnhancedNumberValidator(int)", for a float it would be "EnhancedNumberValidator(float)", etc. These types must match up, otherwise an error will occur.

With the exception of the validatorId XML attribute, all validator XML attribute demonstrated in Figure 9 are optional for an Enhanced Number Validator. For example, a developer could only specify a minimum value of 0 to restrict all values of a parameter to positive values since no maximum was specified. Appropriate defaults will be selected for step and precision if the developer chooses not to specify them.

```
<ParameterList>
  <Parameter name="Double Parameter" value="1.0" type="double"
    docString="A double parameter" validatorId="1" />
    ...Other Parameters ....
  <Validators>
    <Validator type="EnhancedNumberValidator(double)"
      min="0" max="10" precision="2" validatorId="1"/>
  </Validators>
</ParameterList>
```

Figure 9. Example usage of an Enhanced Number Validator

```
<ParameterList>
  <Parameter name="String Parameter" value="Option 1" type="string"
    docString="A string parameter" validatorId="1" />
    ...Other Parameters ....
  <Validators>
    <Validator type="StringValidator" validatorId="1">
      <String value="Option 1"/>
      <String value="Option 2"/>
      <String value="Option 3"/>
    </Validator>
  </Validators>
</ParameterList>
```

Figure 10. Example usage of a String Validator

String Validators

String Validators are a simple yet powerful tool for restricting string input. String input can be a nice alternative to using some standard set of integer values to represent various settings for a parameter. The problem is a simple wrong keystroke can cause big problems. By using a String Validator, an application developer can ensure that a user only selects a string value that is appropriate for a given parameter. String Validators may only be used on parameters of type std::string, otherwise an error will occur.

Figure 10 shows how to use a String Validator. Children ¡String¿ tags with a mandatory "value" XML attribute specify which values are valid. In this example, "Option 1", "Option 2", and "Option 3" will be the only values the user can select from in the GUI.

21
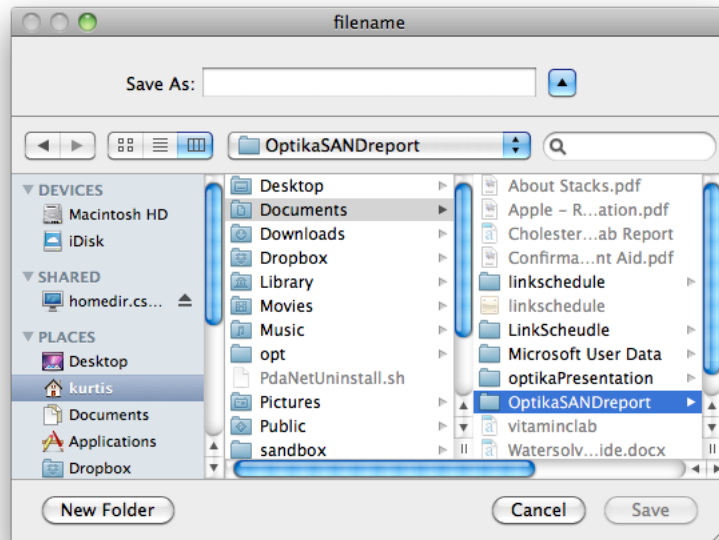
Figure 11. The file selector widget presented to the user when a Filename Validator is applied to a parameter

Filename Validators

Filename Validators allow application developers to designate a string parameter as a filename. When applied to a parameter, any attempt to edit that parameter will result in the user being presented with a special file selector widget (Figure 11). If the "fileMustExist" XML attribute is set to true, the user will only be able select files that already exist. Otherwise, any file may be selected by the user. Filename Validators can only be applied to parameters of type string. If the fileMustExist attribute is not present, it is assumed to be false.

Figure 12 shows an example of how to use a Filename Validator. Here we use it to select a file to output the results of our program. We set the "fileMustExist" XML attribute to false so that the user can select a file that does not already exist. Note that we could've just not include the fileMustExist attribute at all and it would've assumed to have been false.

Array Validators

Array Validators allow any of the above described validators to be used on a array parameter. For example, we might want to use a String Validator on an array of strings. To do this, we create an Array Validator with a "prototype" String Validator. The "prototype" validator is just a term used to describe the validator the developer would like to be applied to each entry of an array. Prototype

```
<ParameterList>
  <Parameter name="Output File Name" value="" type="string"
    docString="Name of the output file." validatorId="1" />
    ...Other Parameters ....
  <Validators>
    <Validator type="FilenameValidator" fileMustExist="false"
      validatorId="1">
  </Validators>
</ParameterList>
```

Figure 12. Example usage of a Filename Validator

validators can be specified in one of two ways. First, the developer can code the validator as child of the array validator tag as in Figure 13. In this case, the prototype validator does not need an Id. Second, the developer can simply specify another validator as an array validator's prototype by using the "prototypeId" XML attribute as in Figure 14. This second option allows the developer to reuse the prototype validator in other instances.

When creating an Array Validator, the "type" XML attribute will include both the type of the prototype validator and the type of the array template parameter. For instance, if we were using a prototype Filename Validator on a string parameter, the value of our "type" XML attribute would be "ArrayValidator(FilenameValidator, string)". It is crucial that all of these types match up. Otherwise, errors will occur.

All the discussion above is also relevant for TwoDArrays. The only difference is instead of using an ArrayValidator, a TwoDArrayValidator is used. For example, if a TwoDArray of arbitrary integer values needs to be restricted with an EnhancedNumberValidator, the XML in Figure 15 would be appropriate.

23

```
<ParameterList>
  <Parameter name="Options" value="{Option 1, Option 1, Option 1}"
    type="Array(string)" docString="An array of options"
    validatorId="1" />
    ...Other Parameters ....
  <Validators>
    <Validator type="ArrayValidator(StringValidator, string)"
      validatorId="1">
      <Validator type="StringValidator">
        <String value="Option 1"/>
        <String value="Option 2"/>
        <String value="Option 3"/>
      </Validator>
    </Validator>
  </Validators>
</ParameterList>
```

Figure 13. Example usage of an Array Validator in which the prototype validator is declared as a child of the array validator

```
<ParameterList>
  <Parameter name="Output File Name" value="{Option 1, Option 1}"
    type="Array(string)"
    docString="Name of output file." validatorId="1" />
    ...Other Parameters ....
  <Validators>
    <Validator type="ArrayValidator(StringValidator, string)"
      validatorId="1" prototypeId="2"/>
    <Validator type="StringValidator" validatorId="2">
      <String value="Option 1"/>
      <String value="Option 2"/>
      <String value="Option 3"/>
    </Validator>
  </Validators>
</ParameterList>
```

Figure 14. Example usage of an Array Validator in which the prototype is specified using the prototypeId XML attribute

```
<ParameterList>
  <Parameter name="Integer Values" value="2x2:{4,3,1,2}"
    type="TwoDArray(int)" docString="A twod array of integers"
    validatorId="1" />
    ...Other Parameters ....
  <Validators>
    <Validator type="TwoDArrayValidator(EnhancedNumberValidator(int), int)"
      validatorId="1">
      <Validator
        type="EnhancedNumberValidator(int)"
        min="0"
        max="10"
      />
      </Validator>
    </Validator>
  </Validators>
</ParameterList>
```

Figure 15. Example usage of a TwoDArrayValidator

Dependencies

Dependencies are one of the most powerful tools provided by Optika [1]. They allow application developers to express common dependencies that occur between parameters in their program. At their core, dependencies allow a developer to say "based on the state of parameter A, parameter B should behave in a certain way." In this case, we would say parameter A is the "dependee" parameter and parameter B is the "dependent" parameter. All dependencies have at least one dependee and one dependent. When running the GUI, the algorithm for expressing dependencies is as follows:

1. A parameter's value is changed by the end-user.

2. The GUI queries the associated defined dependencies to see whether or not the parameter that changed has any dependents.

3. If the parameter does have dependents, the GUI requests a list of all the dependencies in which the changed parameter is a dependee.

4. For each dependency, the "evaluate" function is called. The dependency makes any necessary changes to the dependent parameter(s) and the GUI updates with the new data.

5. If any dependents now have invalid values, focus is given to them and the user is requested to change their value to something more appropriate.

Like validators, dependencies are declared in XML using the ¡Dependencies¿ tag. This tag must be a direct child of the root ¡ParameterList¿ tag. Inside the ¡Dependencies¿ tag each dependency is declared using the ¡Dependency¿ tag. Within the ¡Dependency¿ tag the XML attribute "type" is required in order to define the type of the dependency. Each ¡Dependency¿ must have at least one ¡Dependee¿ child tag and one ¡Dependent¿ child tag. Each of these tags must have an XML attribute called "parameterId" which identifies their associated parameter. A dependency must have an arbitrary number of dependents.

Visual Dependencies

Visual Dependencies allow the application developer to show and hide parameters based on other parameters' values. This is useful in situations when a parameter takes on a particular value, and as a result, another parameter is no longer relevant. By hiding this now irrelevant parameter from the user the application developer can hopefully avoid some confusion on the user's part. If we were to write these dependencies as a sentence, they would say: "Based on the value of the dependee parameter show or hide the dependent parameter(s)".

---

[1]Dependencies are actually found in the Teuchos package. However they were intentionally developed for use in Optika

```
<ParameterList>
  <Parameter name="Favorite Food" type="string" value="pasta"
    id="1" docString="Your favorite food."/>
  <Parameter name="Cheese rating" type="int" value="5"
    docString="A rating of cheese" id="2"
    ...Other Parameters ....
  <Dependencies>

    <Dependency showIf="true" type="StringVisualDependency">
      <Dependee parameterId="1"/>
      <Dependent parameterId="2"/>
      <StringValues>
        <String value="Swiss Cheese"/>
        <String value="Cheddar Cheese"/>
      </StringValues>
    </Dependency>

  </Dependencies>
</ParameterList>
```

Figure 16. Example usage of a String Visual Dependency

Visual Dependencies have an extra boolean XML attribute that can be put in the ¡Dependency¿ tag called "showIf". Which ever dependee is being tested to determine the dependents visibility, showIf can negate it by being set to false. So if a visual dependency is set to only show a dependent when the dependee has a particular value, setting showIf to false will cause the dependent to be visible only when the dependee *does not* have a particular value. If the showIf attribute is not present, it is assumed to be true. With Visual Dependencies, dependents can also be parameter lists in addition to regular parameter dependents.

String Visual Dependencies

String Visual Dependencies allow an application developer to base whether or not a parameter is visible on the particular string value of another parameter. For instance, we might say if the "Favorite Food" parameter is equal to the value "Cheddar Cheese" or "Swiss Cheese" then show parameter "Cheese Rating". Otherwise, do not show parameter "Cheese Rating". Figure 16 demonstrates how we would express this in XML. The "Cheese Rating" parameter will only be shown when "Favorite Food" is set to "Cheddar Cheese" or "Swiss Cheese". String Visual dependencies must have a single dependee of type string.

```
<ParameterList>
  <Parameter name="Use special parameters" type="bool" value="true"
    id="1" docString="Whether or not to use special parameters"/>
  <ParameterList name="Special Parmeters" type="int" id="2"
    docString="special parameters only used in certain instances"
    ...Special Parameters ....
  </ParameterList>
  <Dependencies>


    <Dependency showIf="true" type="BoolVisualDependency">
      <Dependee parameterId="1"/>
      <Dependent parameterId="2"/>
    </Dependency>


  </Dependencies>
</ParameterList>
```

Figure 17. Example usage of a Bool Visual Dependency

Bool Visual Dependencies

Bool Visual Dependencies allow the developer to base the visibility of a parameter on the boolean value of another parameter. For instance, we might say if parameter A is set to true then we want to show parameter list B. Figure 17 shows how a developer would implement this in XML. The "Special Parameters" list will only be shown when "Use special parameters" is set to true. Bool Visual Dependencies must have a single dependee of type bool.

Number Visual Dependencies

Number Visual Dependencies allow the developer to base the visibility of a parameter on the numerical value of another parameter. In their simplest form, Number Visual Dependencies simply check to see if the dependee's value is greater then or equal to zero. If the value is greater than or equal to zero, the dependent(s) will be shown. Otherwise, the dependent(s) will be hidden. Figure 18 shows basic usage of a Number Visual Dependency. Unless the "Temperature" parameter is below zero degrees Celsius, then we cannot have any ice cubes in the room. We do this by setting the "showIf" XML attribute to false so that "Number of ice cubes" only shows when the "Temperature" parameter is below zero.

Notice in Figure 18 that like Enhanced Number Validators, the "type" XML attribute of a Number Visual Dependency has to include the type of the parameter it is evaluating (in this case a double). Number Visual Dependencies may have only one dependee. The type of that dependee must match up with the type specified in the "type" XML attribute used in the Number Visual Dependency tag.

```
<ParameterList>
  <Parameter name="Temperature" type="double" value="1" id="1"
    docString="The temperature in the room in degrees celsius"/>
  <Parameter name="Number of ice cubes" type="int" value="1"
    id="2" docString="The number of ice cubes in the room"/>
  <Dependencies>

    <Dependency
      showIf="false"
      type="NumberVisualDependency(double)"
    >
      <Dependee parameterId="1"/>
      <Dependent parameterId="2"/>
    </Dependency>

  </Dependencies>
</ParameterList>
```

Figure 18. Example usage of a Number Visual Dependency

Sometimes it is necessary to test a number typed parameter using criteria other than whether or not its current value is greater than or equal to zero. This can be accomplished by using a ¡Function¿ tag. A ¡Function¿ tag instructs the dependency to run the value of the dependee through a function first, and then test the result of that function to see if it is greater than or equal to zero. Consider the example in Figure 18, only this time we will specify our temperature is in degrees Fahrenheit. In order to get the proper functionality, we will add a "subtraction function" to the dependency that will subtract 32 from the value of the temperature parameter before it is tested against zero. Figure 19 shows an example of this. We set the operand XML attribute to "32" and the type of the function to "SubtractionFunction(double)"[2] since we want to use a subtraction function on a parameter of type double.

Validator Dependencies

A Validator Dependency is a dependency in which the validator that is in use on the dependee is dependent upon the value of the dependee. Both the type of the dependee and the dependent will vary with each type of Validator Dependency and parameter type of the dependent. Validator Dependencies will always have one dependee. Also, dependent parameters in a Validator Dependency will never be a parameter list. A Validator Dependency can have an arbitrary number of dependents.

---

[2]For a complete list of available functions see Appendix A

```
<ParameterList>
  <Parameter name="Temperature" type="double" value="33" id="1"
    docString="The temperature in the room in degrees farenheit"/>
  <Parameter name="Number of ice cubes" type="int" value="1"
    id="2" docString="The number of ice cubes in the room"/>
  <Dependencies>

    <Dependency
      showIf="false"
      type="NumberVisualDependency(double)"
    >
      <Dependee parameterId="1"/>
      <Dependent parameterId="2"/>
      <Function operand="32" type="SubtractionFunction(double)"/>
    </Dependency>

  </Dependencies>
</ParameterList>
```

Figure 19. Example usage of a Number Visual Dependency using a function

String Validator Dependency

A String Validator Dependency allows the application developer to change the validator used on a dependent based on the string value of the dependee parameter. This is accomplished by mapping values the dependee might take on to corresponding validators. Also, a default validator may be specified. This way, if the dependee takes on a value that has not been mapped to a validator, the default validator can be used. If the dependee takes on a value that is not specified in the values to validator mappings and no default validator is specified, then the dependee(s) will have no validator(s).

A String Validator Dependency must comply with the following rules:

- The dependee must be of type string.

- At least one validator to value mapping must be supplied.

- All of the validators which have values mapped to them must be of the same type.

- If specified, the default validator must be of the same type of validator used in the map.

- The validators in the map and the default validator should be appropriate for the parameter type of the dependent(s).

30

```
<ParameterList>
  <Parameter name="Favorite Food Type" type="string" value="Cheese"
    id="1" docString="The type of your favorite food"/>
  <Parameter name="Favorite Food" type="string" value="American"
    id="2" docString="Your favorite food"/>
  ...Other Parameters...
  <Validators>
    <Validator type="StringValidator" validatorId="1">
      <String value="American"/>
      <String value="Swiss"/>
      <String value="Pepperjack"/>
    </Validator>
    <Validator type="StringValidator" validatorId="2">
      <String value="Lays"/>
      <String value="Ruffles"/>
      <String value="Pringles"/>
    </Validator>
  </Validators>
  <Dependencies>
    <Dependency type="StringValidatorDependency>
      <Dependee parameterId="1"/>
      <Dependent parameterId="2"/>
      <ValuesAndValidators>
        <Pair value="Cheese" validatorId="1"/>
        <Pair value="Chips" validatorId="2"/>
      </ValuesAndValidators>
    </Dependency>
  </Dependencies>
</ParameterList>
```

Figure 20. Example usage of a String Validator Dependency

Figure 20 shows an example of how to use a String Validator Dependency. In this example we have two parameters, "Favorite Food Type" and "Favorite Food". If "Favorite Food Type" is set to "Cheese" we want to restrict the values on "Favorite Food" to only values that are names of cheese. But if "Favorite Food Type" is set to "Chips", we want to restrict the "Favorite Food" values to names of chips. Note that if the user sets the value of "Favorite Food Type" to something else other than "Cheese" or "Chips", there will be no validator on "Favorite Food". This can be remedied by either providing the String Validator Dependency with a default validator or by adding a validator to "Favorite Food Types" which only allows the user to enter in either "Chips" or "Cheese".

Range Validator Dependency

Range Validator Dependencies work very similarly to String Validator Dependencies in that they map validators to specific dependee values. However, instead of mapping string values to

validators, Range Validator Dependencies map ranges of numbers to validators. If the dependee's value falls within one of the mapped ranges, then that range's associated validator is applied to the dependent(s). A range is specified by defining its minimum (inclusive) and its maximum (exclusive) values. Also, like String Validator Dependencies, a default validator can be specified. If the dependees value does not fall with in one of the mapped ranges, the default validator is used. If the dependee takes on a value that is not specified in the values to validator mappings and no default validator is specified, then the dependee(s) will have no validator(s).

A Range Validator Dependency must conform to the following constraints:

- The dependee must be a number type.

- At least one value to validator mapping must be supplied.

- All of the validators specified in the mapping must be of the same type.

- If specified, the default validator must be of the same type as the validators specified in the map.

- The validators specified in the map must be appropriate for the type of the dependent(s).

- Ranges cannot intersect.

Figure 21 shows an example of a Range Validator Dependency. In it, we want to use different validators on the "Fondue Food" if the "Fondue Pot Temperature" is different. If the temperature is greater than or equal to 80 or less than 120, we only want the user to be able to select "Cheese" or "Bread". If it is greater than or equal to 120 but less than 180, we only want the user to be able to pick "Chicken" or "Beef". We associated these ranges with their appropriate values in the ¡RangesAndValidators¿ tag by using ¡Pair¿ tags for each range and validator. Notice that like Enhanced Number Validators and Number Visual Dependencies, the type of the dependee parameter is included in the "type" XML attribute of the dependency. In Figure 21 the dependee's type is double, so the type of the dependency is "RangeValidatorDependency(double)".

Bool Validator Dependency

A Bool Validator Dependency allows the developer to apply one validator to the dependent(s) when the dependee is true, and another validator to the dependent(s) when the dependee is false. You could also specify one validator to be applied when the dependee is true and no validator to be applied when the dependee is false (or vice versa). The dependee of a Bool Validator Dependency must be of type bool. If both a "true" and "false" validator are specified, they must be of the same type. Also, the validator(s) specified must be appropriate for the dependent(s).

Figure 22 shows an example of a Bool Validator Dependency. In this example, we only want a validator to be placed on the "Temperature" parameter if "Has Temperature Constraints" is set to true. Therefore we set the trueValidatorId XML attribute to the id of the validator we want to use and do not specify a validator to be used if the dependee is false.

```
<ParameterList>
  <Parameter name="Fondue Pot Temperature" type="double" value="100"
    id="1" docString="The number of dimensions being used"/>
  <Parameter name="Fondue Food" type="string" value="Cheese"
    id="2" docString="The food we're fonduing"/>
  <Validators>
    <Validator type="StringValidator" validatorId="1">
      <String value="Cheese"/>
      <String value="Bread"/>
    </Validator>
    <Validator type="StringValidator" validatorId="2">
      <String value="Chicken"/>
      <String value="Beef"/>
    </Validator>
  <Validators>
  <Dependencies>
    <Dependency type="RangeValidatorDependency(double)">
      <Dependee parameterId="1"/>
      <Dependent parameterId="2"/>
      <RangesAndValidators>
        <Pair min="80" max="120" validatorId="1"/>
        <Pair min="120" max="180" validatorId="2"/>
      </RangesAndValidators>
    </Dependency>
  </Dependencies>
</ParameterList>
```

Figure 21. Example usage of a Range Validator Dependency

```
<ParameterList>
  <Parameter name="Has Temperature Constraints?" type="bool"
    value="true" id="1"
    docString="Whether or not to use temperature constraints."/>
  <Parameter name=" Temperature" type="double" value="0"
    id="2" docString="The temperature"/>
  <Validators>
    <Validator type="EnhancedNumberValidator(double)" validatorId="1"
      min="0" max="50"/>
  <Validators>
  <Dependencies>
    <Dependency
      type="BoolValidatorDependency"
      trueValidatorId="1"
    >
      <Dependee parameterId="1"/>
      <Dependent parameterId="2"/>
    </Dependency>
  </Dependencies>
</ParameterList>
```

Figure 22. Example usage of a Bool Validator Dependency

Array Modifier Dependencies

Number Array Length Dependencies

Number Array Length Dependencies allow the application developer to have the length of an array parameter determined by the value of another parameter. Number Array Length Dependencies must have a dependee with an integer number type and the dependent(s) must be an array. Unless otherwise specified, the value of the dependee parameter is directly used for the length of the dependent array parameter(s).

Figure 23 shows an example of an Number Array Length Dependency. When the "Number of buckets" parameter changes, the array in the "Amount in each bucket" shrinks and expands accordingly. Note how in the "type" XML attribute for the dependency we had to specify both the type of the dependee and the dependent.

If a developer wanted the length of the array to be based on something other then the direct value of the dependee parameter, the dependee's value could be run through a function. This would be done by adding a ¡Function¿ tag inside the ¡Dependency¿ tag, just like in Figure 19 for the Number Visual Dependency.

```
<ParameterList>
  <Parameter name="Number of buckets" type="int"
    value="2" id="1"
    docString="The number of buckets we have."/>
  <Parameter name="Amount in each bucket"
    type="array(double)" value="{0,0}"
    id="2" docString="The amount of water in each bucket"/>
  <Dependencies>
    <Dependency type="NumberArrayLengthDependency(int, double)">
      <Dependee parameterId="1"/>
      <Dependent parameterId="2"/>
    </Dependency>
  </Dependencies>
</ParameterList>
```

Figure 23. Example usage of a Number Array Length Dependency

TwoD Row Dependencies and TwoD Column Dependencies

TwoD Row Dependencies allow the application developer to have the number or rows in a TwoDArray parameter be determined by the value of another parameter. TwoD Row Dependencies must have a dependent with an integer type and the dependent(s) must be a TwoDArray. Unless otherwise specified, the value of the dependee parameter is directly used for the number of rows in the TwoDArray dependent parameter(s).

Figure 24 shows an example of a TwoD Row Dependency. In this example there is a TwoDArray which specifies two parameter values for each surface in some simulation. In other words, each row in the TwoDArray corresponds to a surface and each column in a row corresponds to a particular parameter value for the surface. This means that as the number of surfaces change, rows need to be added or removed from the TwoDArray accordingly. This is accomplished by using the TwoD Row Dependency. Note how the "type" XML attributes for the dependency correspond to the types of both the dependee and the dependent.

If a developer wanted the number of rows to be based on something other than the direct value of the dependee parameter, the dependee's value could be run through a function. This would be done by adding a ¡Function¿ tag inside the ¡Dependency¿ tag, just like in Figure 19 for the Number Visual Dependency.

In addition to the TwoD Row Dependency, there is also another dependency called a TwoD Column Dependency. It functions in almost exactly the same way as the TwoD Row Dependency except instead of modifying the number of rows in a TwoDArray it modifies the number of columns. Let's modify the example above. Instead of having each row represent a surface, each column will represent a surface. Figure 26 shows how to have the number of columns change,

```
<ParameterList>
  <Parameter name="Number of surfaces"
    type="int" value="2"
    id="1" docString="The number of surfaces in a simulation"/>
  <Parameter name="Params" type="TwoDArray(double)"
    value="2x2:{3,4,3,4}" id="2"
    docString="An array containing parameter values for each surface"/>
  <Dependencies>
    <Dependency type="TwoDRowDependency(int, double)">
      <Dependee parameterId="1"/>
      <Dependent parameterId="2"/>
    </Dependency>
  </Dependencies>
</ParameterList>
```

Figure 24. Example usage of a TwoDRowDependency

rather than the number of rows.

It's also worth noting that there is nothing preventing a developer from using a TwoD Row Dependency and TwoD Column Dependency together. A developer could have a dependent parameter whose number of rows and columns are dependent on two different parameters or even on the same parameter. One instance in which a developer might want to have both the number of rows and columns dependent on a single parameter is when the developer is dealing with a symmetric array. Figure **??** shows an example of this. In this example, the developer is specifying some characteristic of a boundary interaction between two surfaces. Since the boundary interaction characteristic between surface one and surface zero is the same as the boundary interaction characteristic between surface zero and surface one, and the developer doesn't want to specify the interaction between a surface and itself, a symmetrical array is needed. And in this case, the number of rows and columns the array contains is dependent on the same parameter.

```
<ParameterList>
  <Parameter name="Number of surfaces"
    type="int" value="2"
    id="1" docString="The number of surfaces in a simulation"/>
  <Parameter name="Params" type="TwoDArray(double)"
    value="2x2:{3,3,4,4}" id="2"
    docString="An array containing parameter values for each surface"/>
  <Dependencies>
    <Dependency type="TwoDColDependency(int, double)">
      <Dependee parameterId="1"/>
      <Dependent parameterId="2"/>
    </Dependency>
  </Dependencies>
</ParameterList>
```

Figure 25. Example usage of a TwoDColDependency

```
<ParameterList>
  <Parameter name="Number of surfaces"
    type="int" value="3"
    id="1" docString="The number of surfaces in a simulation"/>
  <Parameter name="Boundry Interaction Characteristic" type="TwoDArray(double)"
    value="3x3:sym:{-1,-1,-1,5,-1,-1,4,4,-1}" id="2"
    docString="An array containing parameter values for each surface"/>
  <Dependencies>
    <Dependency type="TwoDRowDependency(int, double)">
      <Dependee parameterId="1"/>
      <Dependent parameterId="2"/>
    </Dependency>
    <Dependency type="TwoDColDependency(int, double)">
      <Dependee parameterId="1"/>
      <Dependent parameterId="2"/>
    </Dependency>
  </Dependencies>
</ParameterList>
```

Figure 26. Example usage of both a TwoDColDependency and
TwoDRowDependency for a symmetric TwoDArray

37

```
<Condition type="NumberCondition(int)" parameterId="3">
  <Function operand="5" type="AdditionFunction"/>
</Condition>
```

Figure 27. Example of a Number Parameter Condition

Condition Visual Dependencies

Up until this point, all Dependencies have had a single dependee. Condition Visual Dependencies are a special type of visual dependencies that allow the application developer to create visual dependencies that have more than one dependee. This is accomplished by using "Conditions". Conditions allow the developer to test the state of multiple parameters and generate a boolean value. If a condition evaluates to true then the dependent(s) of a Condition Visual Dependency will be shown. If the condition evaluates to false, the dependent(s) will be hidden. Like the other Visual Dependencies, these evaluation consequences can be reversed using the "showIf" XML attribute. Before discussing Condition Visual Dependencies, we will first consider the various types of conditions that are available in Optika [3].

Parameter Conditions

Parameter Conditions are the most basic types of conditions. They return true or false based on the evaluation of a single parameter's state.

Number Parameter Conditions

Number Parameter Conditions evaluate the state of a number parameter. If the parameter's value is greater than or equal to zero, the condition evaluates to true. Otherwise, the condition evaluates to false. As with other number based entities in Optika, a function can be added to Number Conditions. The parameter's value will be run through the function and then the return value of the function will be compared to zero.

Figure 27 shows an example of a number condition used to evaluate a parameter with an Id of 3. Like the other number based entities in Optika, the number type of the parameter to evaluate is specified as part of the type name for the condition, in this case int. We have added a ¡Function¿ tag to this condition. The result is that if the parameter's value is greater than or equal to -5, the condition evaluates to true. Otherwise, the condition evaluates to false.

---

[3]Conditions are actually found in the Teuchos Package but were developed with the intention of being used in Optika

```
<Condition type="StringCondition" parameterId="2">
  <Values>
    <String value="option 1"/>
    <String value="option 2"/>
    <String value="option 3"/>
  </Values>
</Condition>
```

Figure 28. Example of a String Parameter Condition

```
<Condition type="BoolCondition" parameterId="5"/>
```

Figure 29. Example of a Bool Parameter Condition

String Parameter Conditions

String Parameter Conditions work by comparing a parameter's string value to a list of given values. If the parameter's value equals one of the listed values, then the condition evaluates to true. Otherwise, the condition evaluates to false. The parameter being tested must be of type string. You may list an arbitrary amount of values to compare the parameter's value against, including listing only a single value.

Figure 28 shows an example of a String Parameter Condition. In it, the value of the parameter with an Id of 2 is compared to the specified values of "option 1", "option 2", and "option 3". If the value of the parameter equals any of these values, the condition will evaluate to true. Otherwise, the condition will evaluate to false.

Bool Parameter Conditions

A Bool Parameter Condition is the simplest of all parameter conditions. It simply looks at a parameter's boolean value and returns that as the evaluation result. The parameter to be evaluated for a Bool Parameter Condition must obviously be of type bool. Figure 29 shows an example of the Bool Parameter Condition. If the parameter with an Id of 5 is currently set to true then the condition will evaluate to true, otherwise the condition will evaluate to false.

```
<Condition type="AndCondition">
  <Condition type="BoolCondition" parameterId="3"/>
  <Condition type="NumberCondition(double)" parameterId="4"/>
</Condition>
```

Figure 30. Example of an And Condition

Boolean Logic Conditions

Boolean logic conditions are really what enable the use of multiple parameters as dependees. Given a group of conditions, a Boolean Logic Condition will return an evaluation result by applying a boolean operator to each condition's evaluation result. The three types of Boolean Logic Conditions are:

- And Condition

- Or Condition

- Equals Condition

Figure 30 shows an example of an And Condition. It works by evaluating the two other condition tags and "anding" their results together. The end result is a condition that only evaluates to true if both the Bool Condition and the Number Condition evaluate to true. In this example we only "and" together two conditions. That said, an arbitrary amount of conditions can have a Boolean Logic Condition applied to them (but there must be at least two conditions to evalute). Also note that the conditions nested within a Boolean Logic Condition can be other Boolean Logic conditions with their own child conditions. There is no limit to the depth of this nesting.

Not Conditions

Not Conditions allow the negation of a single condition. If we modify the code in Figure 29 to look like the code in Figure 31, we get a condition that will only evaluate to true when the boolean parameter with an Id of 5 is false and vice versa.

Using Conditions With Dependencies

Figure 32 shows an example of using conditions within the context of a Condition Visual Dependency. We use an Or Condition to "or" together the results of two Number Conditions. The end result is that "Optional Parameter" will only be shown if either of the Number Conditions evaluate to true. Notice that we still have to explicitly declare the dependees of the Dependency.

```
<Condition type=NotCondition">
  <Condition type="BoolCondition" parameterId="5"/>
</Condition>
```

Figure 31. Example of a Not Condition

```
<ParameterList>
  <Parameter name="Parameter A" type="int" value="3"
    id="1"
    docString="First parameter"/>
  <Parameter name="Parameter B" type="int" value="8"
    id="2"
    docString="First parameter"/>
  <Parameter name="Optional Parameter" "type=string"
    value="cheese"
    id="3"
    docString="optional parameter"/>
  ...Other Parameters...

  <Dependencies>
    <Dependency type="ConditionVisualDependency">
      <Dependee parameterId="1"/>
      <Dependee parameterId="2"/>
      <Dependent parameterId="3"/>
      <Condition type="OrCondition">
        <Condition type="NumberCondition(int)"
          parameterId="1"/>
        <Condition type="NumberCondition(int)"
          parameterId="2"/>
      </Condition>
    </Dependency>
  </Dependencies>

</ParameterList>
```

Figure 32. Example usage of a Condition Visual Dependency

```
void customFunction(RCP<ParameterList> parameterList)
```

Figure 33. The signature all custom functions must have

Other Features of Optika

Custom Functions

In addition to the traditional work flow of just calling the getInput function, Optika also allows for the specification of a "custom function." When a custom function is specified, the GUI does not exit when the user clicks the action button. Instead, the specified function is run. This work flow allows developers to create a program in which users can rapidly change parameters and see the results of those changes. The function specified must have the same signature as the function in Figure 33. The ParameterList RCP that is given in the argument is the current values user has specified. To use a custom function, pass a pointer to the function when calling the getInput function.

Customizing Look And Feel

There are a number of ways to customize the look and feel of an Optika based GUI. All of them involve creating an OptikaGUI object. Once the object is created, a call to the exec function will start the GUI running. The following functions change the appearance of the GUI:

setWindowTitle(const std::string& title) Sets the title of the GUI window Optika generates.

setAboutInfo(const std::string& aboutInfo) When the user clicks on the about menu item, this information will be included.

setActionButtonText(const std::string& text) Sets the text that will be displayed in the action button.

setWindowIcon(const std::string& filePath) This function allows the developer to specify the file path of an image that will be used as the window icon for the Optika GUI.

setStyleSheet(const std::string& filePath) This function allows the developer to specify a Qt Style Sheet[4] that will be used to style the Optika generated GUI.

Figure 34 shows an example of using some of these functions. We create an Optika GUI object, set the title of the window to "My GUI", and set the about info to "This is a GUI I created using

---

[4]For more information on Qt Style Sheets, please see [3]

```
#include "Optika_GUI.hpp"
int main(int argc, char* argv[]){
  using Optika::OptikaGUI;
  using Teuchos::RCP;
  using Teuchos::rcp;
  using Teuchos::ParameterList;

  RCP<ParameterList> usersInput = rcp(new ParameterList);
  OptikaGUI myGUI("inputs.xml", usersInput);
  myGUI.setWindowTitle("My GUI");
  myGUI.setAboutInfo("This is a GUI I created using Optika.");

  myGUI.exec();
  //usersInput now contains what ever the user entered.
  //The developer can do what ever they wish with it.
  return 0;
}
```

Figure 34. Example of customizing the look and feel of an Optika GUI

Optika." We then call the "exec" function. Once that function returns, the "usersInput" parameter list will contain all the values the user specified.

Future Development

Development is largely user driven. A small user base for Optika has developed at Sandia National Laboratories. Some Sandia scientists have suggested that a stand-alone version of Optika used to simply configure parameter lists would be very useful. The ability to undo edits and save a configuration result only contaning the parameter list entries that have changed have also been requested.

Contributions

The initial task for creating a GUI solution for Trilinos was proposed by Dr. Michael Heroux of Sandia National Laboratories. All code for Optika was written by Kurtis Nusbaum of St. John's University. The algorithm for evaluating dependencies was also developed by Kurtis Nusbaum. Roscoe Bartlett of Sandia National Laboratories helped design the mechanisms by which Optika interprets XML. Jim Willenbring of Sandia National Laboratories provided assistance with integrating Optika into the Trilinos Framework. The Trilinos mailing lists were also great sources of information for questions involving integration with the Trilinos Framework and the Teuchos packages.

References

[1] CMake. `http://www.cmake.org/`.

[2] Qt. `http://qt.nokia.com`.

[3] Qt style sheets. `http://doc.trolltech.com/latest/stylesheet.html`.

[4] Qt Widget Gallery. `http://doc.trolltech.com/4.6/gallery.html`.

[5] Teuchos: The Trilinos Tools Package. `http://trilinos.sandia.gov/packages/docs/r10.4/packages/teuchos/doc/html/index.html`.

[6] Roscoe A. Bartlett. Teuchos::RCP beginner's guide. Technical report SAND2004-3268, Sandia National Laboratories, Albuquerque, New Mexico 87185 and Livermore, California 94550, 2010.

[7] Michael A. Heroux, Roscoe A. Bartlett, Vicki E. Howle, Robert J. Hoekstra, Jonathan J. Hu, Tamara G. Kolda, Richard B. Lehoucq, Kevin R. Long, Roger P. Pawlowski, Eric T. Phipps, Andrew G. Salinger, Heidi K. Thornquist, Ray S. Tuminaro, James M. Willenbring, Alan Williams, and Kendall S. Stanley. An overview of the trilinos project. *ACM Trans. Math. Softw.*, 31(3):397–423, 2005.

[8] Tamara K. Locke. Guide to preparing SAND reports. Technical report SAND98-0730, Sandia National Laboratories, Albuquerque, New Mexico 87185 and Livermore, California 94550, May 1998.

A   Functions

The basic layout of an function in xml is as so:

```
<Function operand="value of operand" type="type of function"/>
```

The value for the operand attribute must be a number. The value for the type attribue must be one of the following:

- SubtractionFunction(*numbertype*)

- AdditionFunction(*numbertype*)

- MultiplicationFunction(*numbertype*)

- DivisionFunction(*numbertype*)

This list can be exapanded by creating your own custom function objects that subclass the "SimpleFunctionObject" class in the teuchos package and by creating your own custom xml converter that sublcasses the SimpleFunctionXMLConverter class.

DISTRIBUTION:

1 MS 1318   Kurtis Nusbaum, 01426
1 MS 1320   Dr. Michael Heroux, 01426
1 MS 0899   Technical Library, 9536 (electronic copy)