

OPTIKA: A GUI FRAMEWORK FOR PARAMETERIZED APPLICATIONS

KURTIS L. NUSBAUM* AND DR. MIKE HEROUX†

Abstract. In the field of scientific computing there are many specialized programs designed for specific applications in areas like biology, chemistry, and physics. These applications are often very powerful and extraordinarily useful in their respective domains. However, many suffer from a common problem: a non-intuitive, poorly designed user interface. Many of these programs are homegrown, and the concern of the designer was not ease of use but rather functionality. 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 parameter's values in a fashion that is much more intuitive than the traditional "input decks" used by many 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.

1. Introduction. In the world of scientific computing there is a problem: most software developers are far more concerned with the functionality of their software rather than their user interface. This is understandable given the limited time and pressures of scientific computing environments. And in cases where there are only a few users of a piece of software this type of development is tolerable. However, when a piece of software starts to be used by a wider audience, poor user interface design issues come to the forefront and can greatly hinder further adoption of a particular piece of software. Optika¹ is an attempt to solve this problem in a generic fashion for parameterized scientific applications.

Since developers of scientific applications don't really care about user interfaces, Optika needs to provide a minimal amount of hurdles for developers. Also, the end result needs to be an intuitive user interface that can be easily navigated and utilized regardless of the underlying computations being done.

The purpose of this paper is to discuss the development of the Optika package. In doing so we hope to demonstrate how Optika solved some of the issues associated with developing a generic user interface for scientific applications and provide justification for why we chose particular solutions. We will proceed to discuss Optika development in a semi-chronological fashion.

2. Initial Planning and Development. In Fall of 2008, Dr. Mike Heroux identified a need for the Trilinos Framework² to include some sort of GUI package. Dr. Heroux wanted to give users of the framework the ability to easily generate GUIs for their programs, while still providing a good experience for the end-user. Based on previous GUI work we'd done for the Tramonto project [6], a few initial problems were identified:

- How would the GUI be laid out?
- What GUI framework would be used to build the GUI?
- Different types of parameters require different methods of input. How would the program decide how to obtain input for a particular parameter?
- How would the application developer specify parameters for the GUI to obtain?

*St. John's University, klnusbaum@csbsju.edu

†Sandia National Laboratories, maherou@sandia.gov

¹For more information on Optika, please see its documentation [3]

²Optika is part of the Trilinos Project [7].

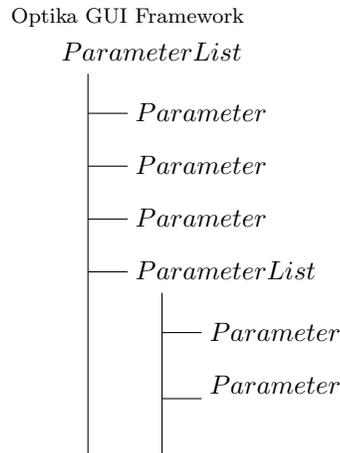


FIG. 2.1. *The hierarchical layout of the GUI*

- How would the application developer specify dependencies between parameters. This was a crucial problem/needed-feature that was identified in previous development of an unsuccessful Tramonto GUI.

After some deliberation, the following initial solutions were decided upon:

- The GUI would be laid out in a hierarchical fashion as shown in Figure 2.1. Parameters would be organized into lists and sublists. This would allow for a clear organization of the parameters as well as intrinsically demonstrate the relationships between them.
- QT³ was chosen as the GUI framework for several reasons:
 - It is cross-platform.
 - It is mature and has a comprehensive set of development tools.
 - It has a rich feature-set.
 - It has been used by Sandia in the past.
 - The Optika lead developer was familiar with it.
- It would be required that all parameters specify their type and the following types would be accepted:

- | | |
|----------|-------------------------------------|
| – int | – string |
| – short | – boolean |
| – float | – arrays of int, short, double, and |
| – double | string |

The supported data type were chosen for two main reasons: (a) The number of input widgets that need to be supported is a direct function of the supported data types. By only supporting a small set of basic data types, we could stick to supporting only a small number of input widgets, most of which were already pre-built and part of Qt. (b) The development team felt that these data types would be adequate for 95% of the developers who would be using Optika.

For number types, a spin box (figure 2) would be used as input. If the valid values for a string type were specified, a combo box (figure 2) would be used. Otherwise a line edit (figure 2) would be used. For booleans, a combo box

³For documentation on all Qt classes please visit [2].

(figure 2) would also be used. For arrays, a pop-up box containing numerous input widgets would be used. The widget type would be determined by the array type (e.g. for numerical types a series of spinboxes would be used).

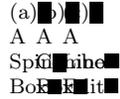


FIG. 2.2. *Some of the various widgets used for editing data [4]*

- Initially it was decided that the application developer would specify parameters via an XML file. A DTD would be created specifying the legal tags and namespaces.
- Dependencies would be handled through special tags in the DTD.

3. Early Development. The first several months of development were spent on creating and implementing the XML specification. The name of the XML specification went through several revisions but was eventually called Dependent Parameter Markup Language (DPML).

After several months of development we realized that creating an entirely new way of specifying parameters might hinder adoption of Optika. We also realized that Trilinos actually had a ParameterList class in the Teuchos [5] package. The ParameterList seemed to be better than DPML for several reasons:

- It was already heavily adopted.
- It had the necessary hierarchical nature (like described in figure 2.1).
- It was serializable to and from XML.

For these reasons, DPML was scrapped in favor of using Teuchos's ParameterLists. Development moved forward with the goal of creating a GUI framework that, in addition to meeting all the challenges outlined above, would also be compatible with any existing program using Teuchos's ParameterLists.

4. Heavy development. Starting in May 2009 a more heavy focus was put on development of the Trilinos GUI package. With the back-end data-structure of the Teuchos ParameterList already in place, attention was turned to developing the actually GUI portions of the framework. A key technology provided by Qt was its Model/View framework [1]. Using the Model/View paradigm, a wrapper class named TreeModel was created around the ParameterList class by subclassing QAbstractItemModel.

However, in subclassing the QAbstractItemModel it was realized that the ParameterList class fell short in terms of providing certain features. The main issue was that a given ParameterEntry located within a ParameterList or a given sublist located within a ParameterList was not aware of its parent. This was an issue because Qt's Model/View framework requires items within a model to be aware of their parents. In order to circumvent this issue the TreeItem class was created. Now the TreeModel class became more than just a simple wrapper class. A TreeModel was created by giving it a ParameterList. It would then read in the ParameterList and create a structure of TreeItems. Each TreeItem then contained a pointer to its corresponding ParameterEntry. This allowed parent-child relationship data to be maintained while still using ParameterLists as the true backend data-structure.

Once the TreeModel and TreeItem class were complete, an appropriate delegate to go between a view and the TreeModel was needed. A new class simply called Delegate

was created to fill this role by subclassing `QItemDelegate`. As specified above, the delegate would return the appropriate editing widget based on the data type carried within a given `TreeWidgetItem`.

With the model and delegate classes in place, an appropriate view could be applied. At first a simple `QTreeView` was applied to the model. Later, as additional functionality was added the view class needed to perform more functions. To fill these needs, the `QTreeView` class was subclassed, creating the `TreeView` class. Its main duties were to show and hide parameters as needed and handle any bad parameter values that might come up during the course of the GUI execution. These features were needed due to requirements that arose from dependencies (something that will be discussed later).

Finally, the `OptikaGUI` class was created. It had one static function, `getInput`. A `ParameterList` is passed to this function, a GUI is generated, and all end-user input is stored in the `ParameterList` that was passed to the function. When the end-user hits the submit button the GUI closes and the `ParameterList` that was passed to the `getInput` function now contains all of the end-user input. The end result was something like that in figure 4.

FIG. 4.1. *The end result of the initial Optika development*

5. Advanced Features. With the basic framework in place, we were now able to move on to more advanced features. As these advanced features were developed various refactorings were made to the already existing code in order to support these new features.

5.1. Validators. One of the goals of Optika is to make life easier for the end-user. It's not enough to simply give the end-user information, it must be conveyed in a meaningful way. Validators are a great way of informing an end-user what the valid set of values for a particular parameter are. Teuchos `ParameterLists` already came with built in validator functionality, but the default validators that were available were sorely lacking in capability. Three initial sets of validators were created to help deal with the short comings of the available validator classes:

EnhancedNumberValidators allowed for validating various number types. `EnhancedNumberValidators` have the following abilities:

- Set min and max.
- Set the step with which the number value is incremented.
- Set the precision with which the number value is displayed.

StringValidator allowed for a parameter to be designated as only accepting values of type string and allowed for specifying a valid list of values.

ArrayValidators allowed for all validator types to be applied to an array of values. The validator that is applied to each entry in the array is called the prototype validator.

A fourth Validator type, a `FileNameValidator`, was added later. This validator designates a particular string parameter as containing a file path and allows the developer to indicate whether or not the file must already exist. Since filenames are such an important type of string, it made sense that they would have their own validator.

By interpreting these validators, Optika could either put certain restrictions on the input widget for a parameter or entirely change the type of input widget used. For instance: with `EnhancedNumberValidators` the min, max, step, and precision of

the EnhancedNumberValidator are all to directly set their corresponding values in the QSpinBox class. But with the FileNameValidator a QFileDialog would appear instead of the normal QComboBox or QTextEdit used for string validators.

5.2. Dependencies. Many times the state of one parameter depends on the state of another. Common inter-parameter dependencies and their requirements include:

Visual Dependencies: One parameter may become meaningless when another parameter takes on a particular value. In this case the end-user no longer needs to be aware of the meaningless parameter and it's best to just remove it from their view entirely so they don't potentially become confused. Visual dependencies should allow the developer to express that "if parameter x takes on a particular value, then don't display parameter y to the end-user anymore."

Validator Dependencies: Sometimes the valid set of values for one parameter changes if another parameter takes on a particular value. Validator Dependencies should allow the developer to express that "if parameter x takes on a particular value, change the validator on parameter y."

Validator Aspect Dependencies: Sometimes the developer doesn't want to change the validator on a particular parameter, but rather just a certain aspect of it. Validator Aspect Dependencies should allow the developer to express that "if parameter x takes on a particular value, change this aspect of the validator on parameter y based on the new value of parameter x"

Array Length Dependencies: Sometimes the length of an array in a parameter changes based on the value of another parameter. Array Length Dependencies should allow the developer to express that "if parameter x changes its value, change the length of the array in parameter y based on the new value of parameter x."

Coming up with a way for the developer to easily express these concepts was not a simple task. The first problem that had to be solved was how to keep track of all the dependencies. They couldn't just be stored in a ParameterList as a class member because of the recursive structure of ParameterLists. Eventually, it was decided that a new data structure called a DependencySheet would hold all the dependencies used for a certain ParameterList. Each Dependency would at minimum specify the dependent parameter and the dependee parameter. However, a complication arose. Because we wanted dependencies to be able to have arbitrary dependents and dependees, we needed a way to uniquely identify the dependee and the dependent. As the ParameterEntry class stood, there was no way of doing this and we didn't want to add this capability to the ParameterEntry class. The Teuchos package is fundamental to the Trilinos Project and before we started changing it for our purposes we wanted Optika to have a solid footing and be absolutely sure that any changes made to Teuchos were actually necessary. We needed to find another way to uniquely identify parameters within a ParameterList.

We decided to use the name of the parameter as the identifier because the accessor functions for a ParameterList usually use the parameters name to identify a particular parameter. While within a ParameterList names of parameters are unique, names are not necessarily unique across a set of sublists. Therefore, in order to uniquely identify a parameter and allow dependencies across sublists Optika would need to know both the parameter name and the parent list containing it⁴.

⁴The astute reader will notice that if there are two sublists with different parent lists and each sublist has a parameter with the same name, then Optika will not be able to uniquely identify the

So it became that every dependency, along with needing the names of the dependee and dependent, also needed their respective parent lists. The `DependencySheet` also needed the root list which contained all of the dependees and dependents. This was so Optika could recursively search for the parameters and their parent sublists (the only way to find them using our method of identification). The following Dependency classes were created to address the use cases above (shown as a hierarchy of classes):

Dependency: Parent class for all Dependencies.

NumberArrayLengthDependency: Changes an array's length.

NumberValidatorAspectDependency<T>: Changes various aspects of an `EnhancedNumberValidator`.

ValidatorDependency: Changes the validator used for particular parameter.

BoolValidatorDependency: Changes the validator used for a particular parameter based on a boolean value.

RangeValidatorDependency<T>: Changes the validator used for a particular parameter based on a number value.

StringValidatorDependency: Changes the validator used for a particular parameter based on a string value.

VisualDependency: Shows or hides a particular parameter.

BoolVisualDependency: Shows or hides a particular parameter based on a boolean value.

NumberVisualDependency<T>: Shows or hides a particular parameter based on a supported number type value.

StringVisualDependency: Shows or hides a particular parameter based on a string value.

Some of these dependencies have fairly novel and robust capabilities. Namely, the `NumberArrayLengthDependency`, `NumberValidatorAspectDependency`, and `NumberVisualDependencies` can all take a pointer to a function as an argument. In the case of the `NumberArrayLengthDependency`, this function can be applied to the value of the dependee parameter. The return value of this function is then used as the length of the array for the dependent parameter. For `NumberValidatorAspectDependencies`, the function is applied to the dependee value and used to calculate the value of the chosen validator aspect. In the `NumberVisualDependency` class, if the function when applied to the dependee value returns a value greater than 0 the dependent is displayed. Otherwise, the dependent is hidden.

The algorithm for expressing dependencies in the GUI is as follows:

1. A parameter's value is changed by the end-user.
2. The `Treemodel` queries the associated dependency sheet to see whether or not the parameter that changed has any dependents.
3. If the parameter does have dependents, the `Treemodel` 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 and the `Treemodel` updates the `Treeview` with the new data.
5. If any dependents now have invalid values, focus is given to them and the end-user is requested to change their value to something more appropriate.

dependent and the dependee. Since solving this problem would most likely require a lot of refactoring of code not directly part of the Optika package, we decided to address it at a later date.

The order in which dependencies are evaluated is arbitrary. We have not tested what happens under the conditions of conflicting dependencies or circular dependencies yet. This is an area for further study.

5.3. Custom Functions. Normally, in Optika the end-user configures the `ParameterList`, hits submit, the GUI disappears, and the program continues with execution. However, an alternative to this work flow was desired. A persistent GUI was needed. The development team added the ability to specify a pointer to a function that would be executed whenever the end-user hit submit. The function was required to have the signature `foo(Teuchos::RCP<const ParameterList> userParameters)`.

5.4. Various Niceties. Various niceties were added to the GUI as well. The ability to save and load `ParameterLists` was added. The Optika GUI class was expanded to allow for customization of the window icon and use of Qt Style Sheets to style the GUI. Checks were also added to see if the end-user was trying to exit the GUI without saving. In such a case they would be warned and given the option to save their work. Tooltips were added so when ever the end-user hovered over a parameter, it's documentation string would be displayed. Also, the ability to search for a parameter was added.

6. Waiting For Copyright. All of the above features were completed around or shortly after the end of August 2009 and Optika was officially given its name. Optika was then submitted for copyright. It took Optika a little over six months to complete copyright. Since it was not yet copyrighted, it could not be included in the Trilinos 10 release in October 2009. During the time Optika spend in copyright limbo, little development on Optika was done. Most of development was cleaning up various pieces of code, adding examples, and adding documentation. Finally, in March 2010 Optika completed copyright and was ready to be included in Trilinos. It was released to the public with the Trilinos 10.2 release.

7. User Feedback. In the summer of 2010, Optika got it's first user. Dr. Laurie Frink began using Optika to create a GUI for Tramonto. There had been a previous attempt to build a GUI for Tramonto, but it had been largely unsuccessful

Initial feedback was very positive. Dr. Frink was very impressed with the capabilities of Optika and the ease at which she could construct a GUI. However, she did have some small initial issues picking up Optika. But most of them arose from the fact she is a C programmer and Optika is C++ based. Her issues were always easily and quickly addressed. Some of her more involved questions even lead to the creation of some great examples.

For the most part Dr. Frink found Optika to be quite adequate for her purposes. However, she did have one rather major feature request: she needed the ability to specify multiple dependents and in some cases even multiple dependees. This was quite a task and required a large reworking of the Dependency part of the framework.

Adding support for multiple dependents was fairly trivial. Instead of specifying a single dependent to the constructor of a `Dependency`, a list of `Parameters` was now passed. If the developer only needed one dependent then he/she could just pass a list of length one. This simple list worked in the case of all the dependents having the same parent list. If they had different parent lists, then a more complex data structure which mapped parameters to parent lists would be used. Convenience constructors were also made for simple cases where just one dependent was needed. The algorithm used for evaluating dependencies changed very little with these modifications. The

only addition needed was an extra loop for evaluating each dependent in a dependency for a given dependee.

Adding support for multiple dependents was much harder. There was actually only one specific use case where multiple dependents were needed or even appropriate for that matter. Dr. Frink needed the ability to test the condition of multiple parameters to determine whether or not a particular parameter should be displayed. So a new `VisualDependency` class called `ConditionVisualDependency` was created. `ConditionVisualDependencies` evaluated a condition object to determine the whether or not a set of dependents should be hidden or shown. The set of condition classes created are as follows (shown as a hierarchy of classes):

Condition : Parent class of all conditions.

ParameterCondition : examines the value of a particular parameter and evaluates to true or false accordingly. Types of `ParameterConditions` include:

BoolCondition: examines boolean parameters.

NumberCondition<T>: examines number parameters.

StringCondition: examines string parameters.

BinaryLogicalCondition: examines the value of two or more conditions passed to it and evaluates to true or false accordingly. Types of `BinaryLogicalConditions` include:

AndCondition: returns the equivalent of performing a logical AND on all conditions passed to it.

EqualsCondition: returns the equivalent of performing a logical EQUALS on all conditions passed to it.

OrCondition: returns the equivalent of performing a logical OR on all conditions passed to it.

NotCondition: examines the value of one condition passed to it and evaluates to the opposite of what ever that condition evaluates.

Through the recursive use of `BinaryLogicalConditions` the developer can now chain together an arbitrary amount of dependents.

`ConditionVisualDependencies` are the only dependencies which allow for multiple dependents. So while support was added for multiple dependents at the `Dependency` parent class level, `ConditionVisualDependency` is the only class which actually implements the functionality. In the case of multiple dependents the algorithm for evaluating dependencies didn't need to change at all.

8. Serialization. The next step was to make Optika's dependencies, conditions, and validators serializable. In addition, we needed to make the parameter serialization more robust. The hope was that by doing this we would allow the application developer to specify the majority of their application via XML, instead of the more cumbersome source-code solution. After some consulting with Roscoe Bartlett, the following model was decided on:

- Each different type of condition, dependency, parameter, and validator would have it's own associated `XMLConverter`. All the dependency converters would sub-class a `DependencyXMLConverter` super-class, all the condition converters would sub-class a `ConditionXMLConverter` super-class, etc. These converters would be able to convert back-and-forth from object and XML.
- A pseudo-database class would be created for each set of converters. All converters would be obtained by calling a `getConverter()` function whose parameter would either be a object to convert, or and XML tag to convert.

The `getConverter()` function would then return an appropriate converter that would allow the object to be converted to and from XML or vice versa.

This model has the benefit of being extendable. If new object types are created, or even if the user desires to create their own types, this model will easily accommodate such additions. A new converter class just has to be created.

The summer of 2010 was spent developing the converters for the standard conditions, dependencies, parameters, and validators. Also, we changed from the system of using a parent `ParameterList` and `Parameter` name to identify specific parameters. We now used pointers, which guaranteed a unique identification of a specific parameter. This was done for two reasons: it was the only good way to make serialization possible, and we determined that it was absolutely necessary for users to be able to uniquely identify parameters. All these changes resulted in a rather large effort that added a lot of new code. In addition, new tests had to be developed for this large addition of functionality. By September we were basically done with only a few loose ends. These were taken care of over the next few months.

9. Future Development. There are several main development goals for Optika in the near future.

- Dr. Frink has submitted several requests for new functionality. Most importantly she has identified the need for users to be able to edit two dimensional arrays. We are currently in the process of adding this functionality. She would also like to be able to specify a label to use for each entry in an array along with a few other nicities.
- We would like to develop a stand-alone version of Optika. The development team believes that the potential audience for Optika is much larger than just the user base of Trilinos. However, creating a stand-alone version presents the problem of keeping source code consistent between the Optika that exists in Trilinos and the stand-alone version. This is an issue that we will need to make sure to address.
- We would like to create a single Optika based executable that acts as a generic `ParameterList` configurator. It would take in a `ParameterList` in XML format, allow the user to configure the `ParameterList`, and then either output the entire `ParameterList` again with the new settings or output a `ParameterList` only containing the parameters that were changed. We think this will be useful because it will enable end-users to utilize Optika without requiring the program they are using to implement Optika support (just `ParameterList` support).
- We need to further investigate what happens when dependencies conflict or become circular. Right now the behavior of Optika under such conditions is unknown.

10. Conclusions. It is difficult to tell if Optika has met its initial goals yet. As of the writing of this paper, Optika only has one user. Hopefully, by continuing to do further development and evangelization its user base can grow. This will then allow us to see if we truly are meeting the needs of the scientific community. Based on early use of Optika by Dr. Frink we believe that Optika is indeed robust enough to meet most of the community's needs but we can't say for sure until we have more user testing.

Appendix A. 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** Reference counted pointer. RCPs referred to in this document reference the RCP class located in the Teuchos [5] 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.

REFERENCES

- [1] *Model/View Programming*. <http://doc.trolltech.com/4.6/model-view-programming.html>.
- [2] *Online Reference Documentation*. <http://doc.trolltech.com>.
- [3] *Optika: Trilinos/packages/optika*. <http://trilinos.sandia.gov/packages/docs/r10.4/packages/optika/doc/html/index.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] *Tramonto Software*. <http://software.sandia.gov/DFTfluids>.
- [7] M. A. HEROUX, R. A. BARTLETT, V. E. HOWLE, R. J. HOEKSTRA, J. J. HU, T. G. KOLDA, R. B. LEHOUCQ, K. R. LONG, R. P. PAWLOWSKI, E. T. PHIPPS, A. G. SALINGER, H. K. THORNQUIST, R. S. TUMINARO, J. M. WILLENBRING, A. WILLIAMS, AND K. S. STANLEY, *An overview of the trilinos project*, ACM Trans. Math. Softw., 31 (2005), pp. 397–423.