Emerging from the MIST: A Connector Tool for Supporting Programming by Non-programmers

Saurabh Bhatia, Tripp Lilley, D. Scott McCrickard
Center for HCI and Department of Computer Science
Virginia Tech
2202 Kraft Drive
Blacksburg VA 24060
{saurabhb,tlilley,mccricks}@vt.edu

Paul Kienzle
NIST Center for Neutron Research
National Institute of Standards and Technology
100 Bureau Drive, MS 8562
Gaithersburg MD 20899
pkienzle@nist.gov

ABSTRACT
Software development is an iterative process. As user requirements emerge software applications must be extended to support the new requirements. Typically, a programmer will add new code to an existing code base of an application to provide a new functionality. Previous research has shown that such extensions are easier when application logic is clearly separated from the user interface logic. Assuming that a programmer is already familiar with the existing code base, the task of writing the new code can be considered to be split into two sub-tasks: writing code for the application logic; that is, the actual functionality of the application; and writing code for the user interface that will expose the functionality to the end user.

The goal of this research is to reduce the effort required to create a user interface once the application logic has been created, toward supporting scientists with minimal programming knowledge to be able to create and modify programs. Using a Model View Controller based architecture, various model components which contain the application logic can be built and extended. The process of creating and extending the views (user interfaces) on these model components is simplified through the use of our Malleable Interactive Software Toolkit (MIST), a tool set an infrastructure intended to simplify the design and extension of dynamically reconfigurable interfaces.

This paper focuses on one tool in the MIST suite, a connector tool that enables the programmer to evolve the user interface as the application logic evolves by connecting related pieces of code together; either through simple drag-and-drop interactions or through the authoring of Python code. The connector tool exemplifies the types of tools in the MIST suite, which we expect will encourage collaborative development of applications by allowing users to integrate various components and minimizing the cost of developing new user interfaces for the combined components.

ACM Classification: H5.2 [Information interfaces and presentation]: User Interfaces. - Graphical user interfaces.

General terms: Design, Management

Keywords: User Interface Design and Management

INTRODUCTION
Software is an evolving concept. Applications go through stages of iterative development in which new functionality is continually added to meet the evolving needs of users. A software product may never be considered complete as it needs regular updates whether in the form of adding new features, fixing existing bugs in the code or just plain making the code more efficient. The traditional approach to development generally involves a customer or a user who has specific requirements from the software and a developer who is capable of programming that software for the customer. Generally there are many people who constitute these two categories and work as a team to achieve a working product.

The constant need for evolving software systems has made it necessary for the end users to perform certain simplistic programming which is often referred to as end-user programming. End-users with little or no software development background have a need to customize or extend an existing application. The benefit of end-user programming is most visible for domain specific applications that target a vertical domain like financial analysis or analyzing scientific data. In this case the end-users are also domain experts who intricately aware of the tacit requirements that their application should meet. Expensive development costs can be avoided by allowing these domain experts to customize and extend their own applications. The lack of suitable tools and infrastructure to support this need for end-user programming is a persistent problem in the software development and interaction design communities. The distinct requirements of diverse groups of users further complicates the issue and makes satisfying all of these needs with a flexible tool a difficult task.

By taking advantage of the unique juxtaposition of a collection of dormant research, contemporary ideas, and a burgeoning body of stable, widespread standards for information exchange we seek to produce an environment in which the initial creation, and long-term maintenance/modification of interactive software carries with it significantly less cognitive overhead than does the present, conventional practice.

This work focuses on development tools that are specifically designed for creating and managing scientific applications. Scientific applications have certain unique features that make them different from other software application domains. Firstly, scientific applications require thorough understanding of the underlying scientific concepts. This means that the scientists who want to use the software are also intimately involved in the development. More often than not, due to budget restrictions, it is the scientists who develop majority of application. While these scientists are experts in their domain they aren't professional programmers and perceive the software development endeavor as an unnecessary hurdle. A scientist programmer would prefer to use an existing code base from an application that was previously written and try to modify it, rather than try and write an entire application from scratch.

Majority of the scientific software involves applications like controlling experimental equipment, capturing data from an experimental setup and analyzing the experimental data. The software application is not just driving the scientific experiment but it is an integral part of the experiment. The experimental parameters could change or the scientists may need to run a different kind of analysis on the data thus requiring changes to be made in the software. Due to the discovery based process inherent to science any scientific software application tends to require constant tinkering around with the application logic or algorithms that drive it. Thus scientific applications constantly evolve in an unstructured manner.

Our area of study is to allow end users with little or no programming experience—the scientists who need the applications—to be able to develop and modify applications themselves, by leveraging and reusing the prior work. We want to let research scientists focus on the scientific computations (e.g., data reduction, filtering, etc.,) and not on programming user interfaces. Scientists can make better use of their time when they are able to dynamically customize their software working environment in response to their evolving data analysis needs. To that end, we are investigating tools, techniques, and paradigms which will make constructing user interfaces more accessible to scientists, and, at the same time, make those user interfaces themselves more open to customization.

The process of extending any existing piece of software can be broken down into two steps: first, understand the existing code and next, write the new code that will extend the application. It is important to understand how the application works, as it currently exists in code, in order to figure out how to extend it. Depending on the level of programming expertise, different individuals use different methods to understand how an existing application works. A professional developer can directly look at the code and gain complete understanding of the application logic. A novice developer may use software visualization tools to gain program understanding (e.g., (Cole, 1989; Stasko, Brown, & Price, 1997; Diehl, 2007)). An end-user builds a mental model of how the application works simply by interacting with the applications interface and as such has no idea about how the application really works in terms of code.

Our target is in building a Malleable Interactive Software Toolkit (MIST), a tool set and infrastructure to simplify the design and construction of dynamically-reconfigurable (malleable) interactive software (Bhatia et al., 2006a; Bhatia et al., 2006b). Malleable software offers the end-user powerful tools to reshape their interactive environment on the fly. Our goal is to make the construction of such software straightforward, and to make reconfiguration of the resulting systems approachable and manageable to a user whose specialty is not in programming but in some other branch of science.

This research effort was designed to integrate with ongoing efforts such as the DANSE (distributed data analysis for neutron scattering experiments) project, which seeks to support scientific research in the neutron scattering field (DANSE, 2009). We believe that these efforts complement those of the DANSE project, and concentrate more on techniques for constructing GUIs which satisfy input/output needs common to scientific programming.

In so doing, we first draw on a diverse body of existing research on alternative approaches to user interface and interactive software construction, including declarative UI languages, constraint-based programming and UI management. We present a model view controller based architecture that provides a foundation for our ideas, and we present a controller tool that enables end users to create connections between two or more existing applications and create new logical constructs thus enabling them to program new functionality.

RELATED WORK

Myers and his colleagues have created the various tools to simplify end user programming. Peridot was one of the initial tools that tried to leverage the power of direct manipulation and programming by example (Myers, 1993). The Garnet project built upon this work to further explore direct-manipulation user interfaces (Vander Zanden et al., 2001). Garnet provided users with a graphical user interface which would enable them to program by simply demonstrating example behaviors on objects. Using the direct manipulation and programming by example techniques allowed users to create graphical interfaces without writing any code. Gamut was perhaps the most powerful tool that

could infer complex behaviors from examples specified by the user (McDaniel & Myers, 1998).

Myers describes the problems that arise out of mixing application logic with user interface code; a problem he refers to as call-back spaghetti (Myers, 1991). This seminal paper proposes the idea of separating code that contains the application logic from the code that describes and controls the user interface. Myers identifies three major problems with call-back spaghetti: Application code gets tied in with an widget toolkit which makes it difficult to port it, incorporating changes and maintaining the user interface becomes difficult and simple modifications like changing the language used in the user interface requires changing the application code itself. Myers goes on to identify the tasks supported by call-backs which force users to mix application logic with user interface code.

The four categories of tasks identified in (Myers, 1991) include: preparing the data for applications which involves changing values that an interface widget returns into values that application code expects, error checking which validates the input data before passing it on to the application, preparing data to be shown to the user which involves setting default values etc. and internal control which defines the connections between user interface components. The paper proposes a system called Gilt which uses filter expressions to minimize the amount of code that needs to be written and in turn avoids the problems of call-back spaghetti. To illustrate the capabilities of the Gilt system the paper describes various examples where Gilt is used to perform the four categories of tasks which were previously identified. The paper concludes with a vision for a future which involves multiple user interface creation and management tools which would work together as an ecosystem to ease the development overhead that is associated with creating graphical user interfaces. This decisive work lays the basis for the separation of concerns between the user interface logic and the application logic; an idea that has gained momentum in the last decade.

In recent years declarative user interfaces have been gaining popularity. Declarative user interfaces examine techniques for separation of user interface logic from the application logic. User interface concerns can be written declaratively as a set of constraints. The constraints are satisfied at runtime usually by a renderer that is also responsible for rendering the declared UI model into an actual UI. Technologies like Mozilla's XML-based User Interface Language (XUL), Microsoft's Extensible Application Markup Language (XAML), and Harmonia's User Interface Markup Language (UIML) are exploring the concept of declarative user interfaces. By allowing high level UI specification, declarative user interfaces are able to separate and isolate the UI code from application code. Declarative UIs show great promise of avoiding the problems of callback spaghetti.

Chin and his colleagues propose the use of domain specific visual approaches to represent scientific models in a computational form which in essence work as user interfaces for the scientific applications (Chin et al., 2006). Existing user interfaces for scientific applications focus on the computational aspects of the application, like data organization and manipulation, rather than the underlying scientific theory. While working with these scientific applications, scientists have to interact with the same WIMP (Windows, Icons, Mouse-Pointer) based interface that most common applications use. Due to this, the domain specific scientific knowledge that is hidden in the program code behind the user interface remains inaccessible to the end user scientists. Graphs and diagrams are useful in conceptualizing scientific concepts, processes and theories. This paper promotes the use of such visual metaphors for the creation of conceptual models which can be linked with application logic to act as a user interface into the scientific work.

The Chin paper describes three specific tools for capturing scientific models: Regional Climate Modeling Problem-Solving Environment (RCM-PSE), Visual Modeling Environment for Biology (VMEB) and Scenario and Knowledge Framework for Analytical Modeling (SKFAM). Regional climate modelers can use RCM-PSE to draw scientific workflow graphs. RCM-PSE helps manage and semi-automate computational experiments by providing mechanisms for capturing, applying and exploring procedural knowledge. VMEB allows biologists to use 2D drawing tools to graphically construct concepts, hypotheses and theories. Using VMEB biologists can use and apply the biological concepts directly, without having to abstract them in terms of the computing domain. SKFAM allows intelligence analysts to create graph-based scenarios of various intelligence cases. These graphical scenarios can also be computationally compared to derive domain specific meaning to the analysis. These tools allow scientists to develop their own user interfaces to computational components while focusing on the conceptual scientific model. This paper claims that the creation of such tools will enable the scientists to take the lead in designing rich and intuitive scientific interfaces.

Wolfgang and his colleagues (2006) introduce the system of user interface façades which allow users to adapt, reconfigure and re-combine existing graphical interfaces. The authors state that the complexity of user interfaces keeps increasing as features are added to the underlying applications. As a typical user is only interested in a small subset of the functionalities provided by an application, exposing all the features in s complex user interface does not make sense. Instead the user interface should be adapted to fit the user's requirements. Façades also allows users to adapt the interaction with the user interface by changing the widgets on the interface along with their mappings to pointer interaction without having to change any code in the underlying application. The paper provides in-depth details of how the façades system was implemented and provides various examples that showcase façades capabilities.

Demeure and his colleagues (2006) describe three representations of software user interfaces that are produced in

the process of software development: a conceptual representation which corresponds to the model or logic that is being used by the software, an internal representation which is the code that formulates the software and an external representation which is the user interface of the software that is seen by end-users. The ubiquitous nature of computing has led to new requirements which demand that a system be adapted to a context of use at run-time. This requirement, which has been termed as plasticity, imposes that the knowledge available during design time also be available to the end-users so that they are able to adapt the system to the current context while preserving the usability of the system. The paper goes on to introduce the concept of a Comet and a related toolset called the Comet Inspector. A Comet consists of a control, a logical abstraction and a logical presentation. A Comet encapsulates alternate logical presentations within itself for different contexts of use. The Comet presentation can be selected automatically or by the end-user thus providing a polymorphic ability to the Comet. Comet Inspector is the tool that allows users to simultaneously view the three representations of the User Interface.

Fujima et. al. propose a system called C3W (Clip, Connect and Clone for the Web) which enables users to create new web based applications by combining elements from existing applications (Fujima et al., 2004). The system is built specifically for HTML based interfaces which provide a form based interface for requesting user input and provide the appropriate output once a user submits the form. The author's describe three problem scenarios which motivated the creation of this system. First, it is difficult for users to locate the input/output elements that interest them inside a webpage if these elements are embedded along with other information. Second, users often have a need to link the results from one form as an input to another. Examples of this include trying to find movies that are currently playing, choosing a movie and then finding out which cinema it is playing in. Finally, users may want to simultaneously compare the results of multiple queries. The C3W system provides integrated support for addressing these issues by introducing the three new interactions of clipping, connecting and cloning.

Systems like C3W, Façades and Comet have been built to solve contemporary problems that exist in the creation and use of user interfaces today. The concepts of plasticity and adaptability of user interfaces lay the groundwork required for the concept of malleability proposed in this work. A Malleable User Interface also allows for changes in the underlying application code which is not supported in plastic or adaptable interfaces.

MODEL VIEW CONTROLLER

The Traits UI package from Enthought is a GUI toolkit that implements a Model-View-Controller based architecture and provides features of user interface plasticity while allowing for easy customization and adaptability of the interface. The Traits defined by the toolkit are wrappers around regular data types like integers and float and have extra metadata associated with them. The metadata is used to create default graphical interfaces to the Traits and to define the interaction behavior. The Traits UI toolkit introspects a model and automatically creates a UI which can then be customized by the user (see Figure 1).

```
class NanoVelcro (HasTraits):
    length = Float(1.0)
    height = Float(1.0)
    distance = Float(1.0)
    angle = Range(0, 360)
```

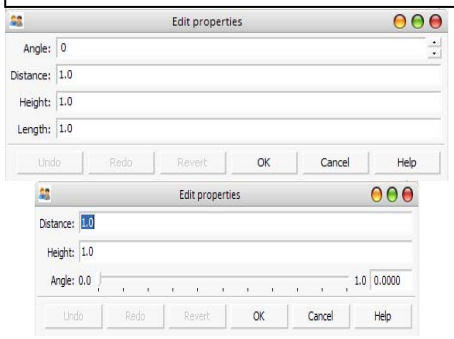


Figure 1: The Traits Model (M) is used to generate a automatic User Interface (V) which can then be customized by the user (V')

The automatic UI generation and the accompanying customization allow users to adapt and customize their UIs while maintaining a live link with the underlying model. Scenarios of using multiple models to create composite interfaces like those supported by C3W are also possible using the Traits UI toolkit. However, the Traits toolkit does not maintain the link between the model and view if the underlying model changes. As shown in (Figure 2) if M0 is the initial model definition then users can use the default view V0 to create their customized views of V0'. If the Model M0 is extended to M1 then a new default view V1 is created and the user may now customize this view to get to V1`. There is no direct path to go from V0' to V1' and all the customizations done by the user in going from V0 to V0' are lost.

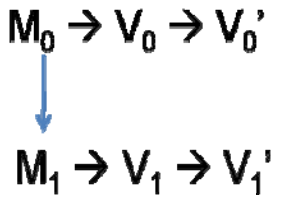


Figure 2: Customizing User Interfaces (Views) using the Traits model

There is a need to provide a path to go from V0' to V1' such that the user customizations in V0' are not lost. Given that application logic has changed, the goal is to reduce the effort required by the end user to incorporate those changes in the UI. The Model View Controller based architecture

simplified the process of building and extending various model components which contain the application logic. The process of creating and extending the views (user interfaces) on these model components can be simplified by using a visualization that will enable comparison of a model component and its corresponding view. Visualization can enable the user to identify parts of the model component that are not present in the view and conversely also help identify user interface components that do not exist in the model. By interacting with the visualization the user can modify the view to account for the changes in the model, thus enabling users to evolve the user interface as the application logic evolves. The visualization can also encourage collaborative development of applications by allowing users to integrate various components and minimizing the cost of developing new user interfaces from the combined components

Version Control for Models and Views

Version control visualizations rely upon version control systems to provide them with effective data. The version control systems are essentially built on the concept of a delta. Whenever a file is modified to create a new version, instead of storing the entire contents of the new file and the old file, the version control systems only store the changes that have occurred between the two versions. This not only saves disk space but allows the user to go back and forth across various versions of the file. This same idea of the delta (Δ) can also be applied to the Traits model to define changes that occur across different models.

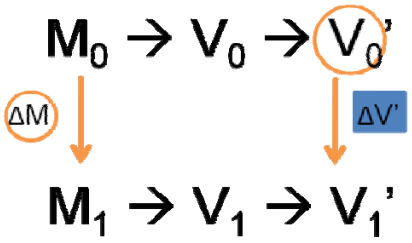


Figure 1: Comparing the changes in model to an existing view

As shown in Figure 1, ΔM is the set of all changes that have occurred between the two Model versions. If this delta is compared with V_0' it will be possible to examine the changes (ΔV') that are needed in V_0' in order to complete the transformation to V_1'

The key to creating an effective visualization is to understand the data it is meant to represent. In this case we need to visualize the changes that have occurred in the model (ΔM) and compare them to an existing view (V_0') (see Figure 1). Since our visualization is focused on models based on the Traits UI framework, it is very easy to enumerate through the changes that can occur when a model is changed. Table 1 summarizes the types of changes that can occur in a model and the corresponding actions that might be needed to link the view with the new model.

Changes to Model (ΔM)	Possible changes to View
Add Trait to Model	Add to View No Change in View
Remove Trait from Model	Remove from View No Change
Change Trait definition	Add to View Remove from View Change View representation No Change

Table 1: Effects of a change in the model on the view

A typical extension of the model will involve a series of such changes. For example, going from M_0 to M_1 could involve Removing Trait_B and Trait_C while Adding Trait_D and changing the definition for Trait_A. Thus using these basic actions we can describe any change (ΔM) that may have occurred in going from one version of the model to the next.

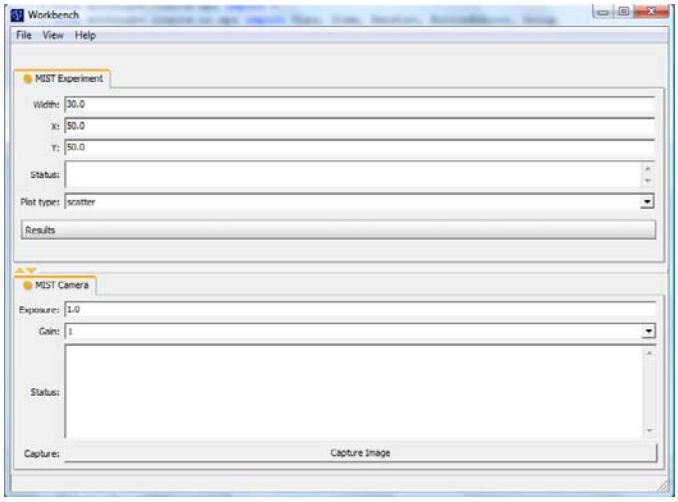
THE CONNECTOR TOOL

The Connector tool enables end users to create connections between two or more existing applications and create new logical constructs thus enabling them to program new functionality, often with simple drag and drop mechanisms. The Connector tool is inspired by the UNIX pipe metaphor which transfers the output of one application as input to another. The Connector tool enables you to create connections by selecting traits as they exist inside a running application and connecting them to a trait inside another application. As the original trait changes value the new value is automatically piped to the receiving trait. The Connector tool is designed to work within the Envisage plug-in framework and works closely with existing tools present in the Enthought Developer Tool Suite.

Envisage is a python package being developed around a plug-in framework motivated by the plug-in architectures present in environments like Eclipse and Netbeans (Chilvers, 2005). Envisage provides a set of standard plug-ins that provide the critical infrastructure required for window management, creating menus and toolbars and other GUI development related tasks. The standard set of plug-ins provides the foundation for creating rich interactive applications.

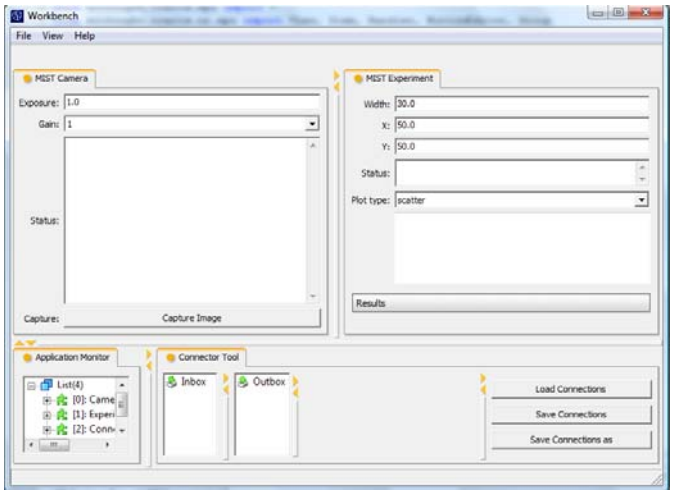
Users can convert any existing traits based python program into an Envisage plug-in by creating a plug-in definition module. A plug-in definition module defines an interface to the plug-in implementation – the actual python class that implements the application. Multiple plug-ins can be combined to work together and create complex extensible applications. For the purposes of this tool we are mainly concerned with the Workbench plug-in which is responsible for the envisage applications user interface. The Envisage framework is still being developed, further details on its

current development status and future plans can be found on the Envisage wiki.



Screenshot 2: An Envisage application consisting of two plug-ins- MIST Experiment and MIST Camera

Envsage's Workbench plug-in provides the framework required to create the user interface for an Envisage application. Any plug-in that has a user interface component must contribute an extension to the workbench plug-in. The workbench plug-in aggregates all its plug-in extensions and creates a single user interface for the envisage application.



Screenshot 2: Multiple plug-ins contribute to the UI for this Envisage application. Each tabbed panel is a plug-in of its own.

To create the user interface the workbench plug-in internally uses the DockWindow sub-package from the Pyface package (enthought.pyface). The Pyface package provides easy access to GUI toolkits like wxWidgets while adhering to the Model-View-Controller design pattern. The DockWindow is a simplified interface provided by Pyface which allows users to quickly create user interface windows. The DockWindow combines a user interface window and a layout manager and provides splitter bars, tabs and drag handles which allow users to rearrange its contents. DockWindows also have the capability to dynamically add new features and functionality to the core DockWindow functionality. This capability to extend the core architecture is called DockWindowFeature. Of particular interest to this work are the Connect and DockControl features which enable some of the core connectivity scenarios implemented by the connector tool.

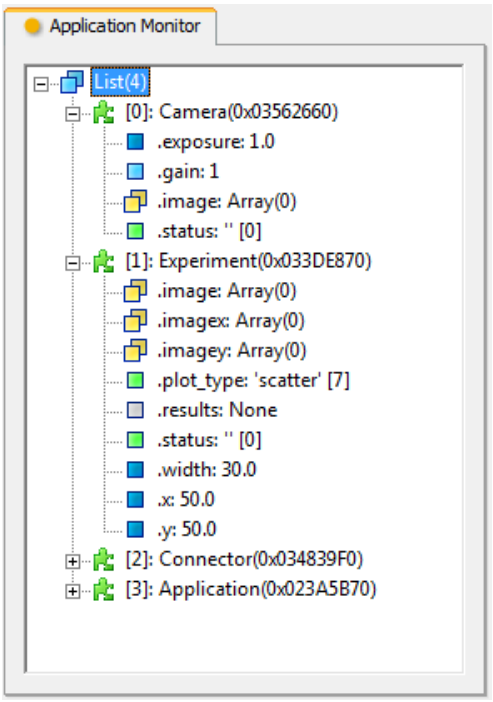
Every DockWindow has a DockSizer object to control the window layout. The DockSizer in turn encapsulates DockControls which display the user interface for the actual python object. The DockControl feature is a way to access an applications DockControl object. Any trait that has the metadata dock_control = True gets associated with the current application's DockControl. All python objects associated with the current application can be accessed through such traits.

The Connector tool we describe in this section builds from the concept of the connect feature. The Connect feature enables users to connect traits that belong to different python application objects. The value of the trait in one application is copied over to the connected trait belonging to another application. Users can thus create their own data flows across multiple applications. In order to make traits inside an application connectable the trait should have an associated connect metadata. The connect metadata can have the values *{to|from|both}*. A *to* value indicates that data may be copied from an application trait to this trait. A *from* value indicates that data may be copied from this trait to any other trait. The both value indicates that data may be copied from and to this trait. Using these values programmers can define how they want various applications to connect with each other. Besides specifying the direction of the connection the connect metadata allows for one optional argument. The optional argument can either be a name or a logical type for the connection trait. The name argument accepts a string that can be used to describe the connection trait. If the name is provided it is displayed to an end user trying to make connections. The type argument works just like the name but additionally requires that any other trait that wants to connect with this trait also have the same string inside its type argument.

The connection feature does allow end user to create customized data flows across applications but it lacks certain important functionality. Firstly, the connect feature requires that the programmer include the connect metadata in the traits while programming the python code. Only the traits which have the connect metadata are displayed to the end user for making connections. Secondly, the connections provide no way performing any filtering or processing before the data is copied over to the connected trait. Lastly there is no easy way of switching between multiple connection configurations. Every connection has to be created or broken individually. We try to overcome these shortcomings while adding powerful new features with the Connector tool.

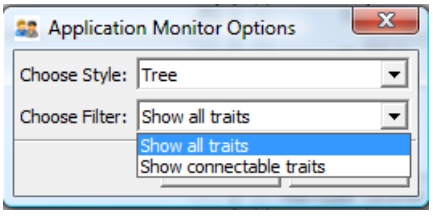
The first problem the Connector tool addresses is to allow end users create connections using any trait present in a running application. To enable this, the Connector tool works together with the Application Monitor, part of the Enthought Developer Tool Suite.

The Enthought Developer Tool suite provides various Envisage plug-ins intended for use by developers creating and debugging Envisage based applications. The developer plug-ins heavily use the DockWindow features present in Pyface. The Application Monitor plug-in displays all the traits that are present in a Python object that is contained by the Envisage application. To be displayed in the Application Monitor the Python object should be associated with a View or Editor that can be displayed inside the Envisage Workbench Window.



Screenshot 3: The Application Monitor plug-in

By displaying all the traits belonging to a python object the Application Monitor effectively enables users to peer into a running application and interact with individual traits inside the application. The default Application Monitor simply displays all the trait names and their current value. This default version was adapted to make it better suited for creating connections with the Connector tool.



Screenshot 4: Filtering options in the modified Application Monitor

A filtering option was added to the Application Monitor to filter the type of traits displayed. By selecting the *"show connectable traits"* option the Application Monitor will only display traits that have the connect metadata. This filtering will display the exact same list as that displayed by the Connect feature. The motivation behind implementing this filtering was to reduce the number of traits that were visible to the user and thus reducing the time spent in looking for the right one. The filtering option does replicate some of the functionality of the Connection feature. However, it does so while providing the rich connectivity options made available through the Connector tool.

The Connector tool provides a central interface for creating and managing multiple connections between various applications. The tools interface consists of an inbox and an outbox. Users can drag any trait that belongs to a live object from the application monitor and drop it into the inbox or outbox. These boxes act as metaphors for collecting all the traits used to create the connections. The inbox contains the traits that function as the source of the connection and the outbox holds all traits that act as the destination. To create a simple connection a user must first drop the trait that is supposed to be the source of the data into the inbox. Similarly the destination trait is dropped into the outbox. To create the connection the user can click on the outbox trait and select the appropriate inbox trait from a drop down list. Alternatively the user can drag and drop the inbox trait onto the outbox trait. Once a connection is made the inbox trait appears as the child of the outbox trait in the outbox tree hierarchy. When a trait is added to the inbox, the connector tool adds a trait change handler to the original trait which is called whenever the trait changes value. This handler performs the function of synchronizing the data across various connections.

To change a connection the user can simply drop another trait from the inbox onto the outbox trait. Alternatively the user can also choose another trait from the "Connect to" drop down. Traits can be deleted from both the boxes by right clicking on them and selecting delete. A warning is displayed if a user tries to delete an inbox trait that is part of an existing connection.

USING THE CONNECTOR TOOL

The Connector Tool provides a programming interface that should be accessible to non-programmers through simple drag-and-drop actions, yet with sufficient power so that a user can learn to extend functionality to support more complex actions. This section provides both a simple and complex example.

The simplest form of a connection is when the value of one input trait is copied over to the value of an output trait. We present the combination of two sample application plug-ins to illustrate this simple connection. One application will gives you the area of a circle given its radius while another application gives you the circumference of a circle given its

radius. The goal is to connect the radius trait of both applications so that it changes together and a user gets both the area and circumference of the circle by simply changing one value. In the end, both applications should be working together as one.

The area calculator has a slider to select the radius. The circumference calculator has a text box where the user can enter the radius. The two behave independently of each other and the goal is to connect the two together. The application monitor shows both plug-ins and their corresponding traits. The radius trait of the area calculator needs to be connected to the radius trait of the circumference calculator.

A user can drag the radius of the area calculator from the application monitor and drop it onto the Inbox of the connector tool. The radius trait will now act as an input source and every time its value changes the connector tool will update the values of all other traits that are connected to it. The radius trait of the circumference calculator should receive the value of the radius trait from the area calculator. In other words the radius trait of the circumference calculator is the output of the connection. To specify this output the radius trait of the circumference calculator is dragged from the application monitor and dropped into the outbox of the connector tool. Now that the input and output traits needed for the connection are specified in the connection tool users' can create an connection by dragging the input radius trait from the inbox and dropping it over the output radius trait in the outbox. A connection has now been made and whenever the slider on the area calculator is used to change the value of the radius the value of the radius trait in the circumference calculator will also change giving the area and circumference of the radius specified by the slider.

One issue still exists though; if you change the value of the radius in the textbox of the circumference calculator the circumference calculator updates the circumference to reflect the new value but the area calculator does not update the area. This occurs because we have only created a one-way connection between the radius of the area calculator to the radius of the circumference calculator. Any changes in the radius of the circumference calculator are not being transferred back to the area calculator. To enable a two connection between both applications another connection must be created with the radius trait of the circumference calculator acting as an input and the radius trait of the area calculator acting as the output. Once both connections are created you can change the value of the radius in either place and the area and circumference values will be updated accordingly to reflect the new value of the radius.

Besides simple drag drop creation of connections the Connector also allows advanced users to create complex connections. Users familiar with programming in python can write a python script inside the evaluate field associated with an outbox trait. The evaluate field acts like a mini shell which executes the script written inside it using Python's exec command. The script is executed whenever any trait present in the inbox changes its value. The script executes inside its own namespace but any item currently in the inbox can be referenced by the script. The inbox items can be referenced by using their IDs as displayed in the inbox (e.g. the radius trait belonging to the area calculator plug-in is referred to using AreaCalculator.radius). The keyword 'this' is used to refer to the outbox trait associated with the script. Thus the evaluate script enables the creation of complex connections wherein the data from one or more input traits can be collected and processed before being pushed out to an output trait.

Complex connections can be very useful in scenarios where the unit of a particular field needs to be changed before it can be used for another calculation:

```
Example Evaluate Script
if 'tmp' not in dir():
    tmp =0
tmp = tmp + 1
this = "%s Image %s\n" % \
 (Camera.status[:-1],tmp)
```

The evaluate script also enables creating connections that loop back to the same trait. This means that data from an input trait can be processed and pushed back to the same trait. The connector tool also allows persisting variables across the script executions: the local namespace is preserved across script initializations so that you can define variables in the script that hold their value as long as the envisage session is active.

CONCLUSIONS AND FUTURE DIRECTIONS

Constructing software is easier than it has been in the past, as is constructing a graphical user interface. However, constructing extensible, maintainable software—software that is robust and interactive—is still a very difficult task. This paper describes our steps toward a Malleable Interactive Software Toolkit (MIST), a tool set and infrastructure to simplify the design and construction of dynamically-reconfigurable (malleable) interactive software. Malleable software offers the end-user powerful tools to reshape their interactive environment on the fly. Our goal is to make the construction of such software straightforward, and to make reconfiguration of the resulting systems approachable and manageable to a user whose specialty is not in programming but in some other branch of science.

This paper presents a diverse body of existing research on alternative approaches to user interface and interactive software construction, including declarative UI languages, constraint-based programming and UI management. We describe a model view controller based architecture that provides a foundation for our ideas, and we present a Controller tool that enables end users to create connections between two or more existing applications and create new logical constructs thus enabling them to program new functionality.

Our expectation is that there should be extreme decoupling among artifacts, where one can express as much as possible in terms of declarative bindings (e.g., RDF/XML, Notation-3, UIML, XAML, or something similar). In so doing, interaction is creation of state change, and constraint enforcement is response to state change (also leading to further state change). By building on the Traits and Traits UI best practices, we expect that it should be possible to create direct-manipulation tools to generate output that is easy to understand and easy to maintain.

ACKNOWLEDGEMENTS

Thanks to members of the Cal Tech DANSE researchers for their input on this work, in particular Michael Aivazis. Thanks also to the researchers at NIST who provided valuable input and coordinated visits, including Przemek Klosowski and Wenwu Chen. And thanks to the Virginia Tech researchers who were involved in early discussions of this work, including Chris North, Dennis Neale, and Brian Sciacchitano.

REFERENCES

Bhatia, S., McCrickard, D. S., Lilley, T., North, C., and Kienzle, P. (2006). Scientists in the MIST: Simplifying Interface Design for End Users. Poster paper in *Proceedings of the World Conference on Educational Multimedia/Hypermedia and Educational Telecommunications (ED-MEDIA '06)*, Orlando FL, June 2006, pp. 653-657.

Bhatia, S., McCrickard, D. S., Lilley, T., North, C., and Kienzle, P. (2006). Scientists in the MIST: Simplifying Interface Design for End Users. Technical Report TR-06-13, Computer Science, Virginia Tech, 2006.

Chilvers, M. (2005). Envisage—An Extensible Application Framework. In *Proceedings of PyCon 2005*, Washington DC, March 2005.

Chin, G. Stephan, E.G., Gracio, D.K., Kuchar, O.A., Whitney, P.D., and Schuchardt, K.L. Developing Concept-Based User Interfaces for Scientific Computing. *Computer 39* (2006) pp. 26-34.

Cole, W. G. Understanding Bayesian Reasoning Via Graphical Displays. In *Proceedings of CHI'89 Human Factors in Computing Systems* (April 30–May 4, Austin, TX), ACM/SIGCHI, NY, 1989, pp. 381–386.

DANSE: Distributed Data Analysis for Neutron Scattering Experiments. Available at http://wiki.cacr.caltech.edu/danse/index.php/Main_Page Accessed June 23, 2009.

Demeure, A., Calvary, G., Coutaz, J., and Vanderdonckt, J. The Comets Inspector: Manipulating Multiple User Interface Representations Simultaneously. In *Proceedings of the 6th International Conference on Computer-Aided Design of User Interfaces (CADUI'2006*), Springer-Verlag, Bucharest, 2006, pp. 167-174.

Diehl, S. (2007). *Software Visualization - Visualizing the Structure, Behavoiur, and Evolution of Software*. Springer, 2007.

Fujima, J. Lunzer, A., Hornb, K., and Tanaka, Y. Clip, connect, clone: combining application elements to build custom interfaces for information access. In *Proceedings of the 17th annual ACM symposium on User interface software and technology (UIST 2004)*, ACM Press, Santa Fe, NM, USA, 2004, pp. 175-184.

Gary, M.R. (1972). Optimal binary identification procedures. *SIAM J. Appl. Math. 23*, 2 (Feb. 1972), 173–186.

Garey, M.R. and Johnson, D.S. (1979). *Computers and Intractability: A Guide to the Theory of NP- Completeness.* Freeman, San Francisco, CA, 1979.

Maloney, J.H., Smith, R.B. (1995). Directness and liveness in the morphic user interface construction environment. In *Proceedings of the 8th Annual ACM Symposium on User interface Software and Technology (UIST 1995),* Pittsburgh, Pennsylvania, United States. ACM Press, New York, NY, pp. 21-28.

McDaniel, R. and Myers, B. (1998). Building applications using only demonstration. In *Proceedings of the International Conference on Intelligent User Interfaces (IUI 1998)*, January 1998, San Francisco CA, pp. 109-116.

Myers, B. (1991). Separating application code from toolkits: eliminating the spaghetti of call-backs. In *Proceedings of the 4th annual ACM symposium on User interface software and technology (UIST 1991)*, ACM Press, Hilton Head, South Carolina, United States, 1991, pp. 211-220.

Myers, B. (1993). Peridot: Creating User Interfaces by Demonstration. In *Watch What I Do: Programming by Demonstration.* A. Cypher et al., eds. Cambridge MA: The MIT Press, pp. 125-153.

Stasko, J. T., Brown, M. H., & Price, B. A. (1997). *Software Visualization*: MIT Press.

Vander Zanden. B.T., Halterman. R., Myers, B. A., McDaniel R., Miller. R., Szekely. P., Giuse. D.A., Kosbie. D. (2001). Lessons learned about one-way, dataflow constraints in the Garnet and Amulet graphical toolkits. *ACM Transactions on Programming Languages and Systems 23* (6), pp. 776-796.

Wolfgang, S., Olivier, C., Dusty, P., and Nicolas, R. (2006). User interface façades: towards fully adaptable user interfaces. In *Proceedings of the 19th annual ACM Symposium on User Interface Software and Technology (UIST 2006)*, ACM Press, Montreux, Switzerland, 2006, pp. 309-318.