Comput Visual Sci (2013) 16:181–192
DOI 10.1007/s00791-014-0230-y

Visual reflection library: a framework for declarative GUI programming on the Java platform

Michael Hoffer · Christian Poliwoda · Gabriel Wittum

Received: 4 August 2011 / Accepted: 16 January 2012 / Published online: 17 December 2014

Abstract The automated mapping of program functionality to intuitive user interfaces is a highly challenging task. Nevertheless it is a promising way to significantly improve software quality by simplifying the development process. This paper describes a method for a declarative and fully automated creation of graphical user interfaces from Java objects, i.e. the information accessible via the Java Reflection API. For this purpose we created the Visual Reflection Library (VRL). VRL interfaces are able to represent complex workflows and allow for a certain degree of visual programming. We start by describing an application: the development of an interactive user interface for the simulation system UG. By shortly discussing the requirements for such an interface, we will explain the reasons for creating VRL and the benefits we gained from it. After that we give an overview of our methods and show several applications. We end by summarizing our results and giving a future outlook.

1 Introduction

In the last decades, increasing computing power made more and more accurate simulations possible. Simulations used to design industrial products solve coupled problems, e.g., detailed spatially resolved computations of turbulent flow in combustion engines, coupling it with combustion models and thermomechanics. Usually these kind of simulations require the simultaneous control of numerous input parameters, like the full geometry, including material properties, coupling of different physical processes, different grids, numerical methods, simulation software and finally computers. Such a simulation process may contain incompatible components [2]. In particular, controlling such an involved process and synchronizing it with the data requirements is very hard to achieve. That means, that simulation complexity nowadays does not only refer to the mere computational complexity of e.g. solving systems of linear equations, but extends to controlling the complex and involved workflow of the simulation process itself. Thus, it is substantial to develop and establish tools allowing easy control of the simulation workflow.

The first step is establishing a graphic and intuitive environment for the user. There are several types of users. An important one is the expert in the application problem to be simulated, but not in numerics or computing. Other users have a lot of expertise in numerics as well. Thus, we need a flexible environment, which allows to set up a simulation in close analogy to the decisions and selections the user is making when approaching the simulation of a model.

As an example for this, we use the simulation system Unstructured Grids (UG), a general platform for the numerical solution of partial differential equations [10]. This simulation system is being developed in the last author's group since more than 20 years. It is a flexible and powerful simulation system, allowing to solve pde-based models from very different application areas and to couple them. In its current state it allows interaction via a built-in scripting language and a corresponding shell. This enables detailed and flexible system control. However, the complexity of the system requires particular skills in programming. The time necessary to learn

Communicated by: Randolph E. Bank.

M. Hoffer (✉) · C. Poliwoda · G. Wittum
Goethe-Center for Scientific Computing (G-CSC),
Goethe Universität Frankfurt am Main, Kettenhofweg 139,
60325 Frankfurt am Main, Germany
e-mail: michael.hoffer@gcsc.uni-frankfurt.de

C. Poliwoda
e-mail: christian.poliwoda@gcsc.uni-frankfurt.de

G. Wittum
e-mail: wittum@gcsc.uni-frankfurt.de

and understand the scripting language, such that someone is able to set up and run a new application independently varies around several months. Further, the scripting language describes the methods used and the necessary parameters in a flat, i.e., chronological way. This was suitable when this language was developed, but for simulations with a complicated workflow this is not appropriate any more. Instead, we need a new tool which allows a hierarchical approach to set up a simulation, corresponding to the way a typical user would approach it. Since there are different expert levels in different areas (application, numerics, computing, …), the tool must offer flexibility in selecting and specifying models, methods and parameters.

To overcome those problems, we decided to create an interface that is able to reflect the complexity and flexibility of UG. As UG is not a regular application, but a platform for the development of numerical applications, we had several requirements for the interface. Instead of creating a separate interface for each UG based application, we tried to find a general solution. Firstly, the creation of intuitive user interfaces is rather time consuming. Changing the program often implies changes in the implementation of the interface. Secondly, it is almost impossible to prevent program errors introduced by the interface. This increases the time and effort necessary for testing. Thus, we wanted a platform for automatic GUI (graphical user interface) generation that is capable of creating user interfaces only by supplying the required functionality of the interface. The Visual Reflection Library [7] addresses this problem. Another requirement is the support for domain specific programming, i.e., domain specific GUI elements. By combining visual programming and problem specific interfaces, it is possible to build an efficient and intuitive platform that fulfills these requirements.

2 Declarative GUI development

2.1 Definition

Declarative GUI programming is already used by several technologies such as Qt [12] and JavaFX [14]. Those toolkits introduce specific scripting languages for defining custom GUI elements. While making the development of custom user interfaces more efficient, this approach does not solve the problem that the connection between the non-graphical backend of the application and the frontend (GUI) has to be created manually and changed whenever the application structure changes.

Declarative Programming as used by VRL is focused on the last aspect. We succeeded in creating a mechanism that allows to keep the backend and frontend in synchronization. VRL does not introduce a specific language for

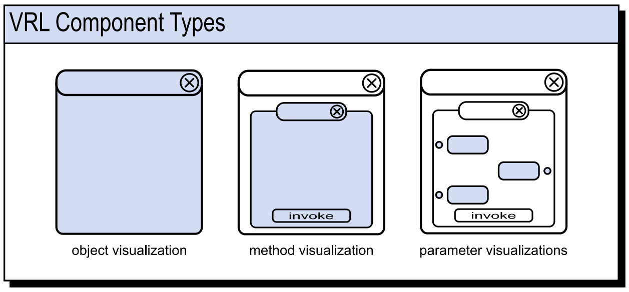


Fig. 1 VRL component types

GUI development. Rather than that, we automatically generate interactive visualizations of Java [13] objects, i.e., their public interface. This totally decouples the GUI generation from the language, i.e., objects from every language that can be accessed via the Java Reflection API can be visualized. Throughout this paper we use Groovy[1] code unless noted otherwise.

As mentioned before, VRL GUI generation is based on the Java Reflection API. There are other tools and frameworks that make use of this information as well to create user interfaces. A common use of this is a property editor as used in development environments such as the Netbeans IDE [15] or Eclipse [4]. But a property editor is optimized for manipulating data and does not give interactive access to the functionality of an object. The BlueJ IDE [9] provides interactive access to the functionality of an object. The intention here is to give an introduction to object-oriented programming. This is an attempt to allow visual interaction with Java objects. However, it does not provide a full mapping from functionality to graphical interfaces.

These usages of GUI generation via Reflection do not solve the problem of automatically generating high-quality interfaces. The challenge is to create a general-purpose framework for declarative GUI programming.

2.2 VRL component types

To accomplish this task, VRL uses three types of visual components (see Fig. 1). An object representation is a container, comparable to a program window that can group several child components. A method representation is a container component inside an object representation. It can also group child components and provides elements for calling the represented method. To represent variable data VRL provides type representations. In most cases they allow interaction with the visualized data.

[1] Groovy is a dynamic language for the Java platform (see: [3]).

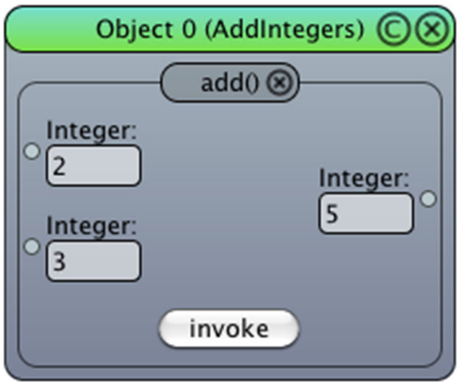


Fig. 2 Visualization of a simple Java object

2.3 Object visualization

Figure 2 shows the visualization result for a simple class that provides a method that is capable of adding two integer numbers. Based on the component types of Sect.2.2 it is possible to create interfaces with only a minimal amount of code:

Listing 1 Code used to create the visualization shown in Fig. 2

```
public class AddIntegers {
  public Integer add(Integer a, Integer b) {
    return a+b
  }
}
```

To create a graphical user interface it is only necessary to specify the functionality. In our case we request a GUI that is capable of adding two integer values. Compared to Java code that implements this functionality and also involves GUI generation, this example shows that declarative GUI development can be very efficient. It has to be mentioned that this example visualization has several extra features such as error messages for incorrect data etc. For a common Java implementation using the Swing [11] toolkit much more code is required. A sample implementation[2] is shown in Listing 2.

Furthermore, VRL can do the interface generation without any additional interface related commands. But by defining the problem domain, interfaces can be customized (see Sect. 2.4).

This is one of the major advantages. The VRL user only provides functionality and optionally some details about the problem domain. These details are not commands. They are meta information, i.e., annotations[3] that do not influence the functionality of the given code. The absence of GUI related commands keeps the code simple and clean. Even more important is the fact that the code can be used for other purposes as well. That is, the implemented functionality can be used in any Java program or library without changing the code. For source code that does contain GUI related commands other than annotations this is in often impossible.

Even though architectural patterns such as the Model-View-Control pattern (MVC) [5] improve the development process and help to separate the functionality from the graphical interface, we think that the application logic should be completely independent from the user interface and should not be a part of the implementation. Namely, if not automatically generated, the graphical user interface and the application logic will always diverge to some degree and a lot of work has to be done to prevent that. In many projects this results in user interfaces that cannot provide the newest functionality of the application backend. In these cases users have to disregard the graphical user interface and use the backend directly (programmatically). This adds even more features to the backend that are not accessible through the user interface.

Under the assumption that UG provides a service that is able to create such an interface description for its functionality, VRL visualization can be done without an extra implementation. For the upcoming UG version such a service is in preparation. Our early tests show that this is a highly promising approach.

2.4 Domain specific GUI elements

While object representations and method representations are mostly generic elements, type representations are individual components. Their appearance depends on the data type of the visualized variable and the problem domain. Let us recall the example (Listing 1) from Sect. 2.3. The code does not include any GUI related commands. Thus, VRL uses a default type representation for the parameter types. For numbers and strings this is relatively easy.

Although this seems to be a reasonable approach for simple objects, it is not clear that this is true for complex cases as well. Compared to classical GUI development this seems rather inflexible. To overcome those problems VRL supports multiple type representations per data type. By supplying information about the problem domain, i.e., the context of the variable to visualize, it is possible to advise the system to choose between different type representations. For our example we might want to use a slider instead of an edit field. VRL supports such meta information via parameter annotations. Parameter annotations can be used to request a specific visualization and to define problem specific properties such as value ranges. If the requested visualization is available it will be preferred over the default visualization. Listing 3 shows a customized version of Listing 1. The resulting visualization is shown in Fig. 3.

[2] This is not necessarily the shortest possible implementation.

[3] Annotations are meta-data that can be added to source code without affecting its functionality. See http://download.oracle.com/javase/1.5.0/docs/guide/language/annotations.html for detailed explanations.

Listing 2 Compact implementation of a Swing application that is able to add two numbers

```
public class Main {
  public static void main(String[] args) {
    javax.swing.JFrame frame = new javax.swing.JFrame(``Add Integers'');
    frame.setLayout(new java.awt.GridLayout());
    final javax.swing.JTextField input1 = new javax.swing.JTextField();
    final javax.swing.JTextField input2 = new javax.swing.JTextField();
    final javax.swing.JTextField output = new javax.swing.JTextField();
    frame.add(new javax.swing.JLabel(``Integer''));
    frame.add(input1);
    frame.add(new javax.swing.JLabel(``Integer''));
    frame.add(input2);
    frame.add(new javax.swing.JLabel(``Result''));
    frame.add(output);
    javax.swing.JButton btn = new javax.swing.JButton(``invoke'');
    btn.addActionListener(new java.awt.event.ActionListener() {
      public void actionPerformed(java.awt.event.ActionEvent e) {
        output.setText(new Integer(new Integer(input1.getText()) +
           new Integer(input2.getText())).toString());
      }
    });
    frame.add(btn);
    frame.pack();
    frame.setVisible(true);
    }
}
```

Listing 3 Illustration of parameter annotations

```
public class AddIntegers {
  public Integer add(@ParamInfo(style=``slider'',options=``min=0;max=100'') Integer a,
                     @ParamInfo(style=``slider'',options=``min=0;max=100'') Integer b) {
    return a+b;
  }
}
```

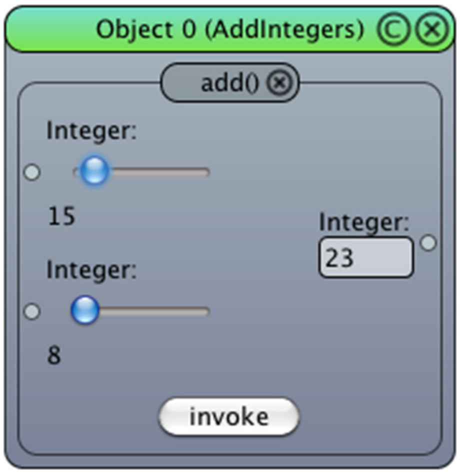


Fig. 3 Visualization of a simple Java object using parameter annotations

Defining properties of the type representation may be impractical in some cases because the annotations will become rather complicated. In this case it is suggested to define a custom type representation which will be discussed in Sect. 2.5.

2.5 Custom type representations

To extend the number of known problem domains, VRL can be extended by adding custom type representations. Currently type representations for common variable types such as `Integer`, `Float`, `Boolean` and `String` exist. In addition it provides interactive type representations for 2D and 3D visualizations. The UG specific extensions enable UG script generation and include MathML [16] based rendering for mathematical formulas (see Sects. 4.1 and 4.2). Extensions for modifying surface properties such as boundary conditions are in development.

Technically a type representation is a Swing component that provides additional methods for data processing and visualization including data specific error handling. Defining custom type representations is a problem specific task.

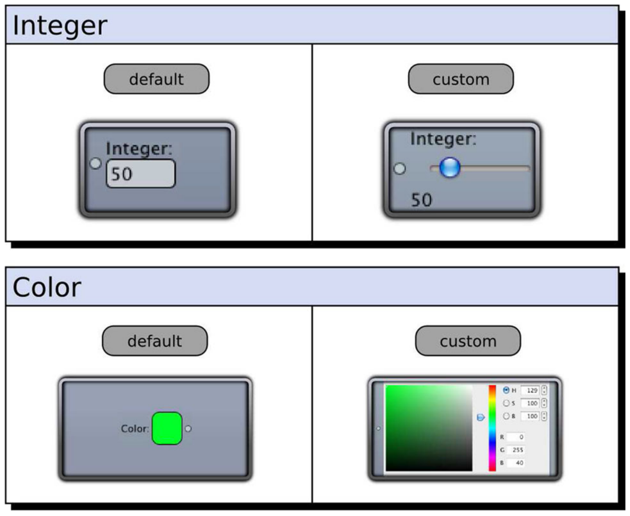


Fig. 4 Type representations for `Java.lang.Integer` and `java.awt.Color`

Listing 4 Circle class

```
class Circle {

    public int radius

    public Circle(Integer radius) {
        this.radius = radius
    }
}
```

Thus, except from basic understanding of the Swing framework, only problem specific knowledge is required.

An overview of different type representations for `java.lang.Integer` and `java.awt.Color` is shown in Fig. 4.

We now discuss these features with the help of an example. Listing 4 shows a sample class we want to provide a type representation for.

If VRL visualizes an object that uses the Circle class from Listing 4, it uses a default type representation that only shows the class name. Defining a custom type representation enables VRL to use an improved visualization.

Listing 5 shows a possible implementation of a type representation for the class defined in Listing 4. The constructor defines the class to visualize and defines the name to use (we use an empty name). The `setViewValue( Object o )` method defines how to visualize an object. For every class that uses the `Circle` class in its public interface VRL will use the type representation defined in Listing 5 unless requested otherwise. The `getViewValue()` method can be used to create an object based on the current visualization, e.g., user input. In our case we do not provide an interactive visualization. Thus, the value that is currently visualized will be returned. In many cases it is sufficient to implement a constructor, the `setViewValue( Object o )` and `getViewValue()` methods. Thus, no special knowledge of the internal implementation is required.

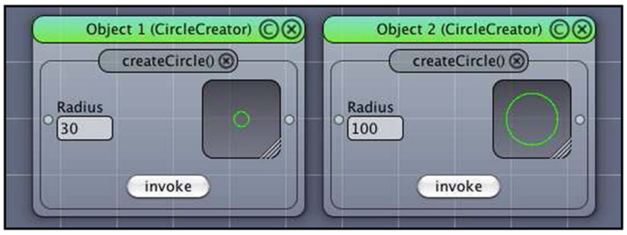


Fig. 5 Class that uses the `Circle` class in its public interface

Listing 6 shows a class that creates an instance of the circle class and returns it. The corresponding VRL visualization is shown in Fig. 5.

3 Visual programming

3.1 Data dependencies

Each visual programming environment needs a method to define data dependencies. Therefore, VRL components can be connected via wires (see Fig. 6). Defining dependencies by connecting components is a technique that is used by several tools such as the Visualization Data Explorer from IBM [8]. We define data dependencies by connecting return values and parameters of methods. VRL connections are typesafe. To evaluate data dependencies it is necessary to define a sequence of method calls, i.e., to determine which methods have to be called to compute the requested result.

Interactive type representations notice every value change. All dependent type representations will be emptied to prevent data inconsistencies. The return value type representations of all dependent methods are marked as out of date. If the user invokes a method VRL will compute the dependencies and call all required methods.

However, determining the sequence of method calls by evaluating the data dependencies limits the user in several ways. It is not possible to freely define a deterministic algorithm. In common programming languages method call sequences can be defined independently from data dependencies.

3.2 Codeblocks

To enable the feature of defining method call sequences, VRL supports codeblocks. A VRL codeblock is equivalent to a block in C++ or Java. VRL contains code generators that can map a sequence of method calls to Groovy code (see Fig. 7). This sequence is defined by selecting method representations via mouse gestures. The corresponding Java object can be visualized like any other object. Hence, VRL provides mech-

Listing 5 type representation for the `Circle` class

```
class CircleType extends BufferedImageType {

    public CircleType(){
        setType(Circle.class)
        setValueName('' '')
    }

    public void setViewValue(Object o) {
        def circle = o as CirclePathProvider::
        def image = ImageUtils.createCompatibleImage(300,300)
        def g2 = image.createGraphics()

        g2.setColor(Color.green)
        def thickness = 5
        g2.setStroke(new BasicStroke(thickness))

        int centerX=image.getWidth()/2
        int centerY=image.getHeight()/2

        int x = centerX + thickness/2 − circle.radius
        int y = centerY + thickness/2 − circle.radius

        int width = 2 * circle.radius–thickness/2 − 1
        int height = 2 * circle.radius–thickness/2 − 1

        g2.drawOval(x,y,width,height)
        g2.dispose()

        super.setViewValue(image)
    }

    public Object getViewValue() {
        return this.@value
    }
}
```

Listing 6 Class that uses the `Circle` class in its public interface

```
class CircleCreator {

    public Circle createCircle(
        @ParamInfo(name=''Radius'') Integer radius) {
        return new Circle(radius)
    }
}
```

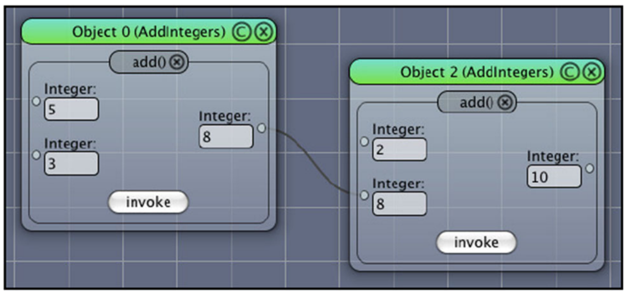
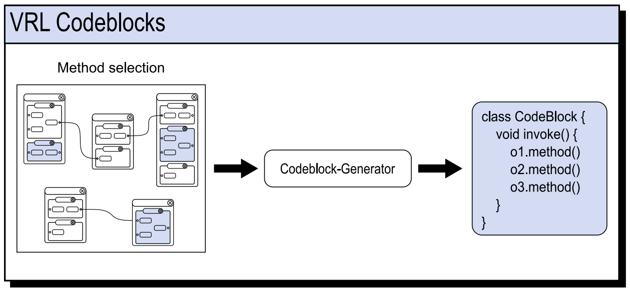


Fig. 6 Data dependency between two objects



```
class CodeBlock {
  void invoke() {
    o1.method()
    o2.method()
    o3.method()
  }
}
```

Fig. 7 Custom sequence of method calls via codeblocks

anisms for defining custom method call sequences. This is important to enable the definition of complex workflows. For UG this feature is essential, as it is necessary to allow custom computation workflows, like the current UG scripting language does.

One problem with the current approach is that it is not easily possible to invoke a method multiple times, each time with different parameters. We addressed this issue by supporting multiple object visualizations. These visualizations can be used to invoke the same method with different parameters.

3.3 IDE features

As the Java VM allows dynamic class loading [6] it is also possible to define the functionality as Java class and visualize it with VRL at run-time. In addition to that VRL provides Groovy support. An integrated editor enables the user to write custom components. Even type representations can be developed at run-time. This allows interactive GUI development and extends the visual programming features. The Groovy code does not have to be added via the editor. It can also be retrieved from a code generator such as the codeblock generator described in Sect. 3.2. To simplify the development process, we created VRL-Studio, a small VRL based IDE. VRL-Studio is also used to create the UG frontend.

3.4 Persistence

VRL uses a XML based persistence model. It stores canvas properties, objects (instances of classes) and the state of their visualization. The source code of classes defined with the Groovy editor is stored as well. Such a configuration is called a *session*.

Most VRL based programs are based on a session file. To ensure that the final program can be deployed, VRL sessions can be exported. The exported file contains the session file and the VRL run-time system, including external dependencies.

3.5 Creating applications

3.5.1 Problem definition

Even though we have seen several features that improve the creation of user interfaces, we have not yet answered the question whether application development is possible. Creating the workflow is done by connecting objects (some of their type representations) and creating codeblocks. But what about the application itself? In many cases it is necessary to create a reduced user interface, designed for a specific purpose. This is what defines the application from a user point of view. VRL provides several features to achieve this (see Sects. 3.5.2 and 3.5.3).

3.5.2 Parameter groups

After defining the workflow of an application we usually want to group the most important parameter visualizations to simplify the user interface. With the methods we have seen so far, the only choice is to change the classes that define the application functionality or to add classes specifically designed to group selected parameters.

But in some cases this is either impossible or impractical as we do not want to change our code only to achieve a specific grouping of object parameters. This leads to code that is only used to create the GUI itself. However, this is exactly what we try to avoid.

Thus, VRL provides a feature called *Parameter Groups*. This means that, parameters from object visualizations can be selected and grouped in a separate window. This is shown in Fig. 8 and 9.

3.5.3 Window groups

The problem now is that we still see all the objects, whether important or not. Therefore VRL allows the definition of *Window Groups*. A window group defines the location and

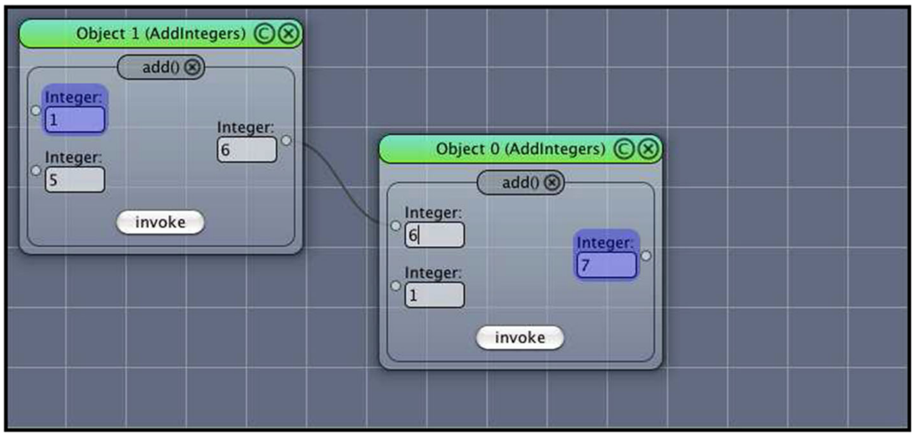
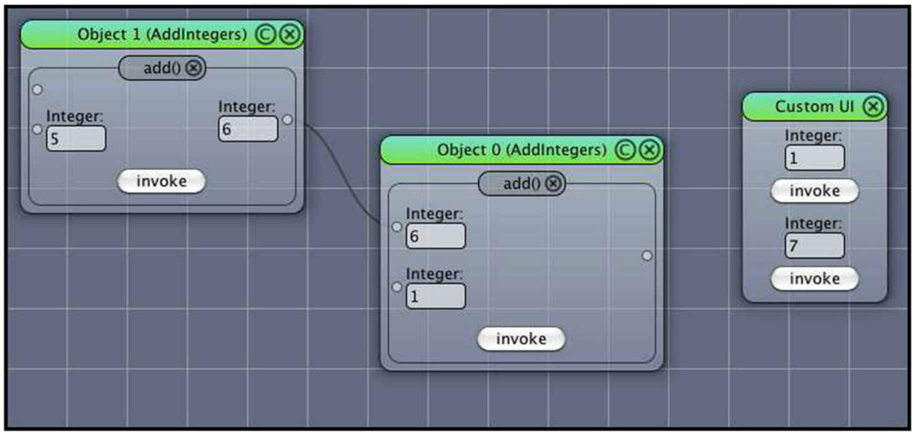
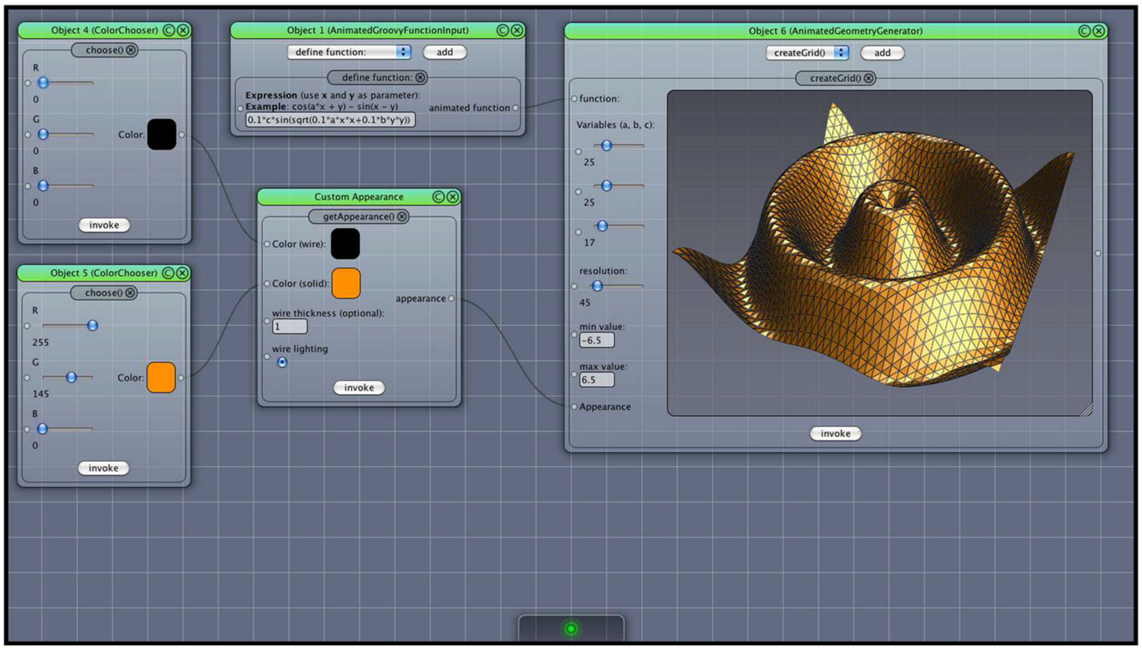


Fig. 8 Parameter groups (Selection)



Object 1 (AddIntegers): Integer: 5, Integer: 6
Object 0 (AddIntegers): Integer: 6, Integer: 1
Custom UI: Integer: 1, Integer: 7

Fig. 9 Parameter groups (Result)



Object 4 (ColorChooser): R 0, G 0, B 0, Color: ■
Object 5 (ColorChooser): R 255, G 145, B 0, Color: 🟧
Expression (use x and y as parameter):
Example: cos(a*x + y) – sin(x – y)
0.1*c*sin(sqrt(0.1*a*x*x+0.1*b*y*y))
Color (wire): ■
Color (solid): 🟧
wire thickness (optional): 1
wire lighting
Variables (a, b, c): 25, 25, 17
resolution: 45
min value: -6.5
max value: 6.5

Fig. 10 Function plotter

the visibility of each window that is part of the group. This enables the user to hide all objects that shall not be part of the reduced user interface.

3.5.4 Example

Combining both features, an application workflow can be simplified significantly. Figure 10 shows a simple function plotter. It provides objects for defining the function that shall be plotted, properties that define the function variables and the visual appearance of the output. Finally, it shows the output itself. It is an interactive 3D visualization of the evaluated function.

We assume that the only important task is the definition of the function and their parameters. The final application interface is shown in Fig. 11.

3.5.5 Multiple views

As we have seen in Sect. 3.5.4, VRL enables the definition of a task specific view. However, it is also possible to define multiple window groups. This enables the definition of multiple task specific views. In the function-plotter-example this could be a separate view for changing the appearance of the visualization.

3.5.6 Limitations

Currently, the definition of a reduced user interface has some limitations. When grouping parameters it is not possible to choose between different component layouts and a parameter cannot be grouped twice. As we think that this is an important feature we plan to integrate advanced layout support in the near future.

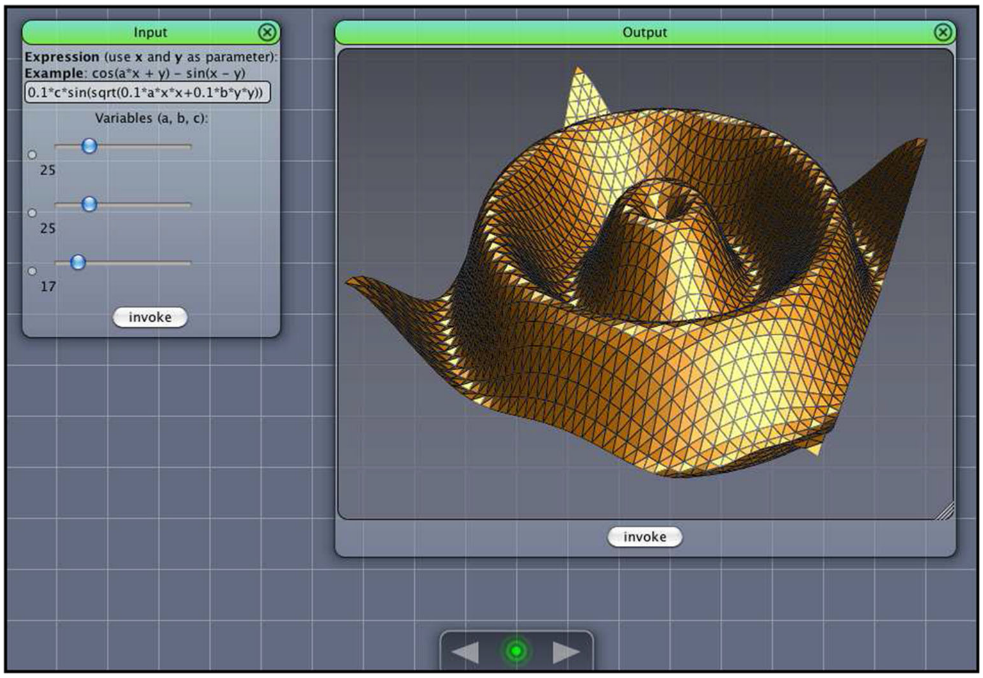

Input
Expression (use x and y as parameter):
Example: cos(a*x + y) – sin(x – y)
0.1*c*sin(sqrt(0.1*a*x*x+0.1*b*y*y))
Variables (a, b, c):
25
25
17
invoke
Output
invoke

Fig. 11 Reduced interface for the function plotter (see Fig. 10)

4 VRL applications

Now we show some real examples to demonstrate the current state of VRL.

4.1 BasicMath

BasicMath is an extension for VRL. BasicMath provides several mathematical objects such as scalar, vector and matrix and corresponding functions such as vector norm or matrix vector multiplication etc. The mathematical objects are represented as visual instances. BasicMath enables visual formulation of mathematical expression. Object names can be displayed as mathematical expressions, i.e., special characters such as integral, norm are supported.

To render mathematical expressions BasicMath integrates the MathML[16] renderer JEuclid [1].

However, it is not necessary to use MathML code for object names. Plain text is also supported. All functions/operators can automatically create the name of their result. By combining several functions/operators the name of the last result consists of the complete expression. But the result name can be overridden with an arbitrary expression.

Functions can be added from a popup menu to the canvas of VRL-Studio. Objects like e.g. a matrix will be created and added to canvas by a so called *MatrixGenerator*.

In addition to Functions/Operators and data elements, BasicMath provides so called generators, one for each element type (matrix, vector, etc.).

We illustrate the usage of generators and functions/operators with the help of the following example.

After creating two vector objects and one matrix object via corresponding generators, we use the objects for calculations. In Fig. 12 objects for vector addition and matrix-vector multiplication are used.

In Sect. 4.3.2 we describe a component that uses the data objects shown in the example to interact with UG. In Fig. 13 we see a scalar, matrix and a vector object. They are used by the component that interacts with UG.

It is possible to visually access data elements in different ways, e.g., the data of a matrix object can be accessed by a vector. An example is the manipulation of the matrix diagonal. To enable this feature, BasicMath uses so called mappings, i.e., bijective mappings between two index sets. Visual changes of the data elements affect all visualizations that use these data elements.

4.2 UG 3

In the following part of this article we want to show which graphical components have been created for the current version of UG. Furthermore, we describe their functionality and how they can be used to simplify the UG workflow. We make the assumption that a common user does not want to implement the mathematical algorithms himself. That is, the user wants to use the existing functionally to solve a specific problem without deeper understanding of the workflow internals. The workflow itself is rather static. But the parameters

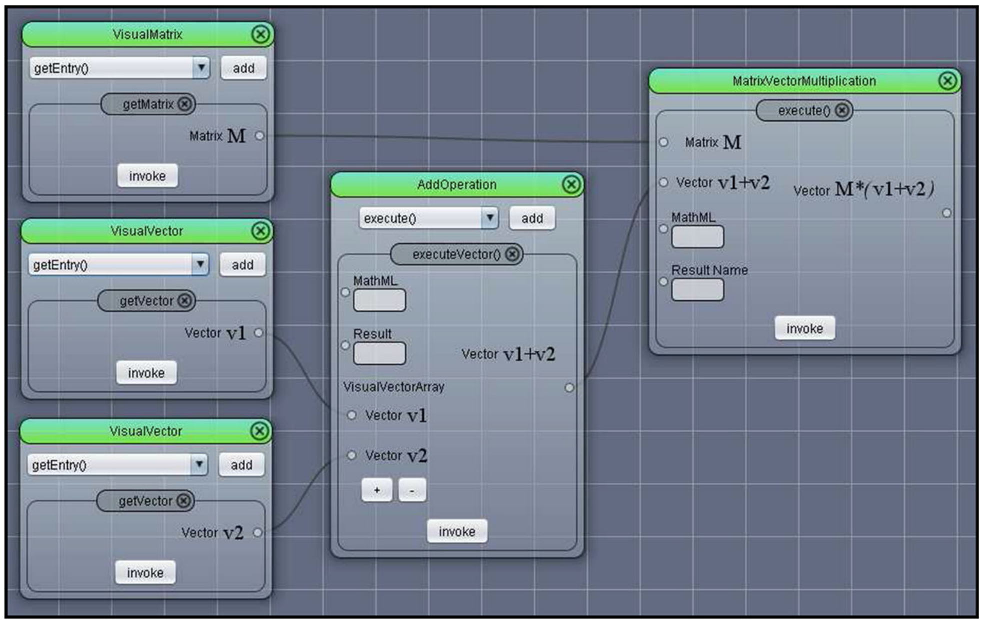
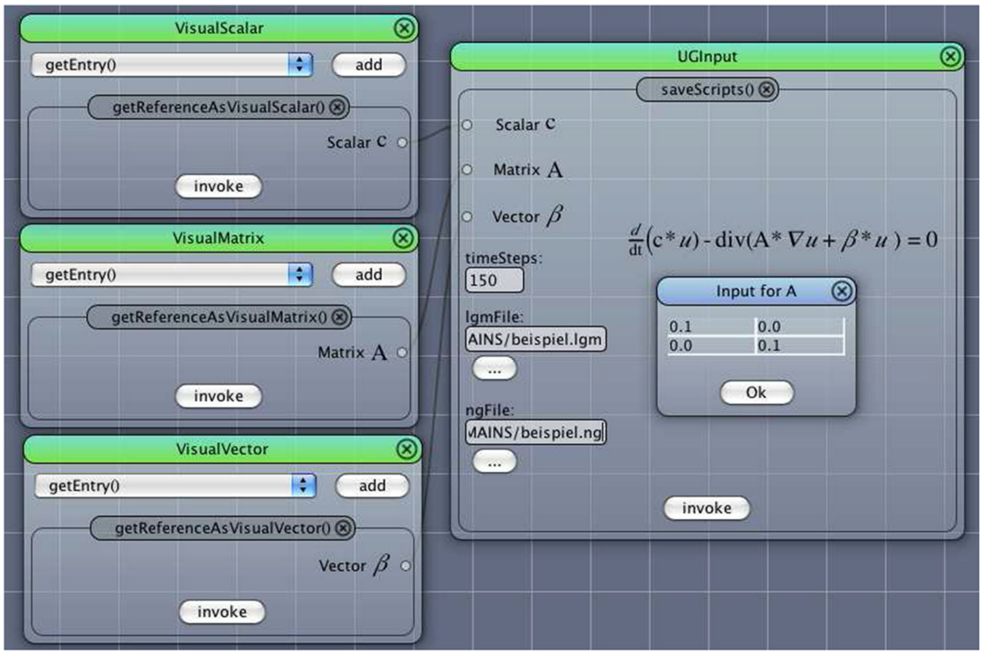


Fig. 12 BasicMath sample session


VisualScalar
getEntry()
add
getReferenceAsVisualScalar()
Scalar c
invoke
VisualMatrix
getReferenceAsVisualMatrix()
Matrix A
VisualVector
getReferenceAsVisualVector()
Vector β
UGInput
saveScripts()
Scalar c
Matrix A
Vector β
$\frac{d}{dt}(c*u) - \mathrm{div}(A*\nabla u + \beta*u) = 0$
Input for A
Ok
...

timeSteps: 150

lgmFile: AINS/beispiel.lgm

ngFile: MAINS/beispiel.ng

0.1	0.0
0.0	0.1

Fig. 13 UGInput with three BasicMath elements

are subject to change and depend on the specific problem. Therefore, all components allow one to interactively specify selected parameters.

Additional type representations and components allow the visualization and interactive manipulation of mathematical objects such as matrices, vectors and scalars. The existence/availability of these visualizations and corresponding data structures enable the visual formulation of mathematical contents/relations/subjects. This functionality is part of the VRL extension BasicMath.

4.3 Line of action

4.3.1 Creating a VRL based graphical frontend

The development process of a VRL module that integrates a specific UG workflow could be classified as follows:

– identify the problem specific parameters
– create custom type representations for special parameter types

– create the Java classes that implement the necessary functionality (will be visualized by VRL)
– create custom script files to store the user-specified parameters
– include the custom script files into the existing UG scripts

4.3.2 Components for the diffusion-convection-equation

To improve the handling of the UG components for the diffusion convection equation

$$\frac{d}{dt}(c * u) - div(A * \nabla u + \beta * u) = 0$$

and the results they create, special components were created.

UGInput shown in Fig. 13 on the right side allows one to set the parameters of the diffusion convection equation and visualizes the equation with user defined parameter names. The equation is based on automatic generated MathML code. This allows a flexible adaption of the visualization of the equation.

Timesteps and geometry can be set by typing the number respectively the path to the desired geometry file.

LivePlotter allows one to observe the evolution of the geometry over time during the calculation. Furthermore, the geometry can be freely translated, rotated and zoomed. With this component we tried to enable the user to evaluate the current solution.

LivePlotter was developed to valuate an intensive calculation in nearly real time, if the evolution of the corresponding geometry is known or estimated to be of a special kind. So the calculation can be aborted if the development did not fit the desired progress.

SolutionPlotter shown in Fig. 14 was created to visualize the evolution of the geometry after the competition is finished. SolutionPlotter allows the same interaction with the geometry as LivePlotter but he can additionally replay the development or visualize the geometry at a specific timestep.

The application area of the SolutionPlotter are e.g. iterative processes where a big interest in the evolution of the geometry and or the visual presentation of it.

PickPlotter shown in Fig. 15 allows different operation states and actions. They are represented by buttons that are shown on the left side of the visualization area.

In the translation-rotation mode it is possible to define a custom rotation point on the geometry surface. When defining the rotation center the geometry will be translated. That is, the rotation point will be moved to the center of the visualization area.

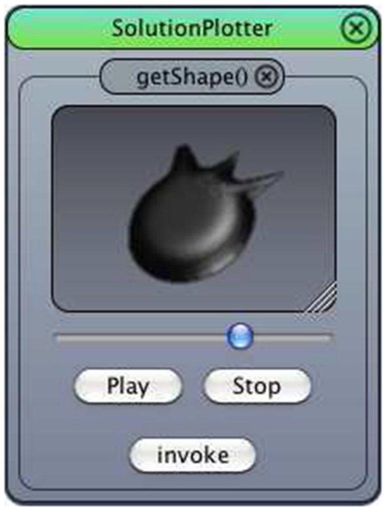


Fig. 14 The SolutionPlotter component visualizing a calculated geometry

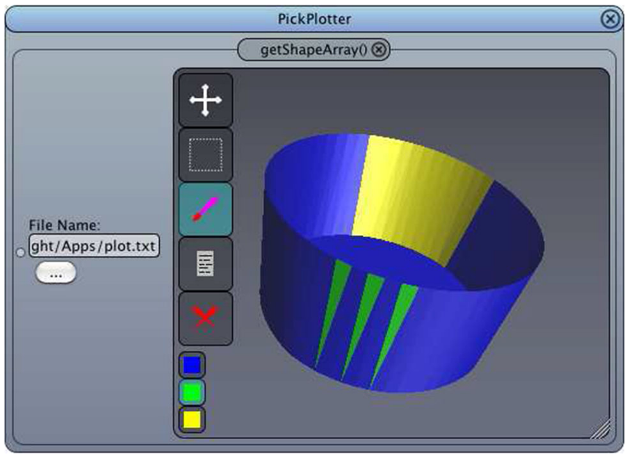


Fig. 15 The PickPlotter component with a loaded geometry and two selected areas

Additionally, there is the possibility to define boundary conditions on the visualized geometry, which are also graphically represented. There is one state to select all triangles along the deep axis in a selection rectangle and one state to select only the first visible triangle under the mouse pointer.

The type of the boundary condition can be selected by pressing one of the three smaller buttons on the left lower corner of the visualization area.

To notify UG about the selection this component can write the custom information to a script file.

4.3.3 Components for the skin-model

The ”skin model” is a diffusion model where the diffusion behavior of medical ointments and cosmetic creams can be observed, which were applied on a piece of human skin.

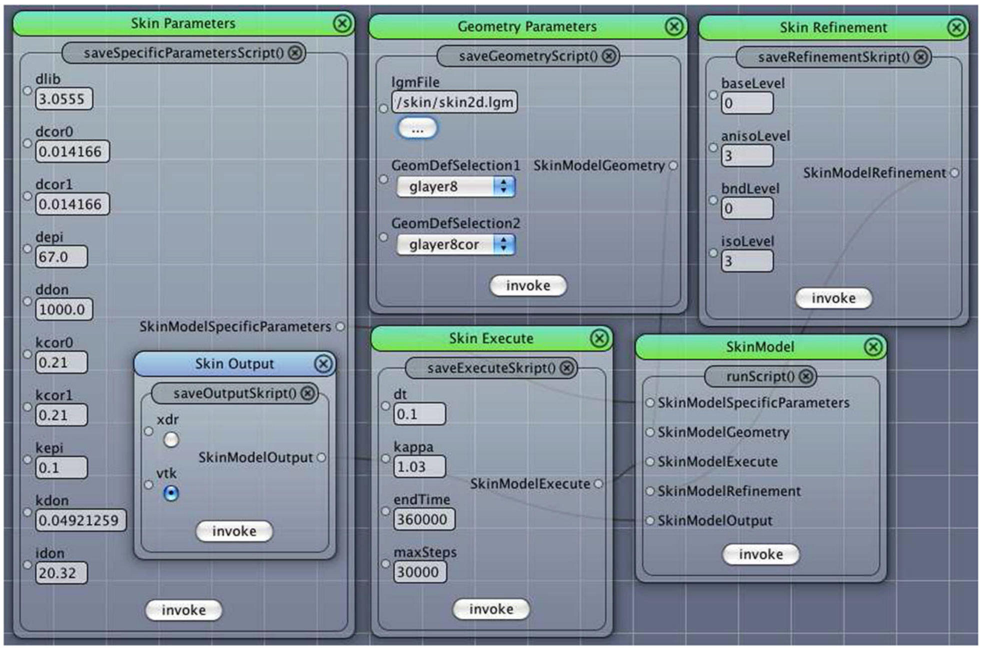


Fig. 16 All components of the user interface for the skin model

Based on the structure of the involved UG scripts which are necessary for the calculation of the skin model we created certain frontends.

All frontends which are necessary for starting the calculation of the skin model are shown in Fig. 16.

5 Conclusion

By directly using the public interface of Java objects it is possible to highly increase the degree of automation in GUI development. This leads to more development productivity and efficiency. Automation is not fully possible for the definition of the problem domain and the visualization types that are to be used. The VRL based extensions contain several new type representations in addition to the already existing ones. Generally, visual programming will be improved by adding new GUI elements that augment the support for drag&drop and simplify the development by custom editor styles.

Our intention is to allow the development of high quality interfaces for simulation tools with minimal effort. We think that this is an important step. It does reduce the probability for program errors and the time-to-discovery. Finally it opens the software to a wider range of users.

Acknowledgments We would like to thank Alexander Heusel, Daniel Jungblut and Alfio Grillo for useful discussions during the preparation of this paper.

References

1. Jeuclid. http://jeuclid.sourceforge.net (2010)
2. ProSTEP iViP Association, D.G.: Integration of simulation and computation in a pdm environment, white paper. (2008).
3. Codehaus Foundation: Groovy 1.7. http://groovy.codehaus.org (2009)
4. Eclipse Foundation: Eclipse 3.6.1. http://www.eclipse.org (2010)
5. Gamma, Erich, Helm, Richard, Johnson, Ralph, Vlissides, John: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Boston (1995)
6. Halloway, S.D.: Component Development for the Java Platform. Addison-Wesley, Boston (2002)
7. Hoffer, M.: Methoden zur visuellen programmierung. Master's thesis, Heidelberg, Univ., Diplomarb. (2009).
8. IBM: Visualization Data Explorer. http://www.opendx.org (1991)
9. Kolling, M., Quig, B., Patterson, A., Rosenberg, J.: The BlueJ system and its pedagogy. Journal of Computer Science Education, Special issue on Learning and Teaching Object Technology **13**(4), 249–268 (2003). http://www.cs.kent.ac.uk/pubs/2003/2190
10. Lang, S., Wittum, G.: Large-scale density-driven flow simulations using parallel unstructured grid adaptation and local multigrid methods. Concurrency - Practice and Experience.
11. Loy, M., Eckstein, R., Wood, D., Elliot, J., Cole, B.: Java Swing, 2nd edn. O'Reilly, California (2002)
12. Nokia, Qt Development Frameworks: Qt 4.6. http://qt.nokia.com (2009)
13. Oracle, formally Sun Microsystems: Java. http://www.oracle.com/us/technologies/java/index.html (1996)
14. Oracle, formally Sun Microsystems: Javafx 1.3. http://javafx.com (2010)
15. Oracle, formally Sun Microsystems: Netbeans ide 6.9.1. http://www.netbeans.com (2010)
16. World Wide Web Consortium: MathML. http://www.w3.org/1999/07/REC-MathML-19990707 (1999)