A Survey of Visual Programming Tools

Dominique Parker
Department of Computing Science
University of Alberta, AB T6G 2E8, Canada
dominiqu@cs.ualberta.ca

July 17, 2003

Abstract

Through their graphical interfaces, visual programming tools provide a considerably different means for specifying programs than traditional text-based programming languages do. Because humans use a lot of imagery while thinking, it is believed that using visual tools is a much more natural approach to programming than writing sequential text. This paper surveys five recent visual programming tools – Create, Prograph, Sanscript, Starlogo and ScriptEase – that have differing purposes and approaches for specifying programs. While these tools have some useful features, they also have their limitations, can be awkward to use, and have yet to revolutionize the art of programming.

1 Introduction

Using visual programming tools to write programs is an alternative to manually writing textual code in a traditional programming language. Visual programming tools come in many forms and are used for various purposes. There are tools for specific tasks such as creating user interfaces and specifying physical simulations, whereas others are for general-purpose programming. Some tools implement pure visual programming languages with programs represented using graphs composed of icons and other visual elements, and the graph is the only form of source code before interpretation or compilation. There are similar tools that convert graphs into textual code of another programming language through a purely mechanical translation. There are other tools yet in which programmers specify their intents using wizards and other conventional GUI widgets.

There has been much research in the area of visual programming tools, because humans think and remember things in terms of pictures. Imagery is an integral part of creative thought. Humans can absorb data much easier from well-defined plots than they can from large tables of numbers, or textual descriptions. Proponents of visual programming therefore argue that the development of visual tools is a natural step in the evolution of programming.

This paper surveys five recent visual programming tools to gain some useful insights into the pros and cons of these tools. Sections 2 through 6 each provide discussion on a different tool. The discussion for each tool describes how the user goes about performing common tasks with the tool, and includes comments on how easy it is to perform those tasks. These sections are presented in the order that the tools were reviewed. Section 7 discusses some lessons learned from using the tools. Finally, Section 8 presents some conclusions.

2 Create

Create, by Sharper Software [1], is a visual programming tool that represents code using flowcharts, otherwise known as control-flow diagrams. It then converts flowcharts into C or Java source code, or into a Windows executable. While flowcharts can be converted into object-oriented Java code, the programming model used to build the flowcharts is procedural.

A free demo for this tool is available from the web-site. As an aside, the quality of the web-site's design and content is extremely poor. The main page only has a vague diagram depicting the interactions that occur between the different parts of the tool, and a hyperlink to *Enter the Site* that absolutely requires a Flash plugin. The hyperlink leads to a Flash introduction that takes longer than necessary to load and contains nothing but Sharper Software's logo and yet another button labeled *Enter*. Clicking on this button finally leads to the content of the web-site except that everything is disabled; everything except a button labeled *Start* that enables everything else once pressed. The

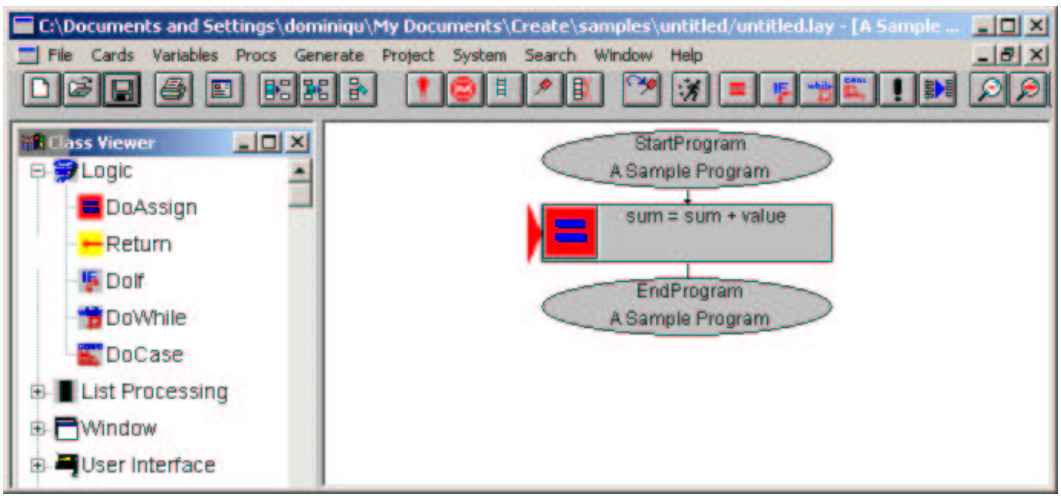

Figure 1: A trivial program in Create.

web-site's content, for the most part, is uninteresting. It is all marketing hype to convince people that their tool is revolutionary and makes programming a simple task, but offers no support for their claims. Some parts of content are just plain confusing, like the following sentence for example:

> Logic structures involving and Variable assignments are color-coded and class objects – which can be drag-and-dropped from CREATE's Class Viewer into the flowchart – have their own unique symbolic icons and family color.

The user interface can be seen in Figure 1. It has an area on the right where the program's flowchart is edited, and on the left is a window called the *Class Viewer*. The Class Viewer's name is misleading since it does not contain any classes; at least not in the object-oriented sense of the word. It just contains a list of all of the different graph nodes representing functions and control-flow constructs that can be added to flowcharts. These are divided up into categories. Nodes are added to the graph by dragging them from the Class Viewer to the flowchart. The node will be inserted into the flowchart directly below the node with the red arrow pointing to it (see the central node in the figure). As the user drags the mouse, the arrow points to the node nearest to the mouse cursor. When a node is attached to a flowchart, a dialog, called a *Large View* immediately appears in which the node is configured. A node's Large View can be opened at any time by double-clicking it.

A program always has two nodes labeled *Start Program* and *End Program* that appear at the extremities of the program's flowchart. The Large View of the Start Program node allows some application-specific details to be supplied, such as the program's name, window dimensions and positioning information. Indeed, Create assumes that all applications are GUI-based. Some quirks of

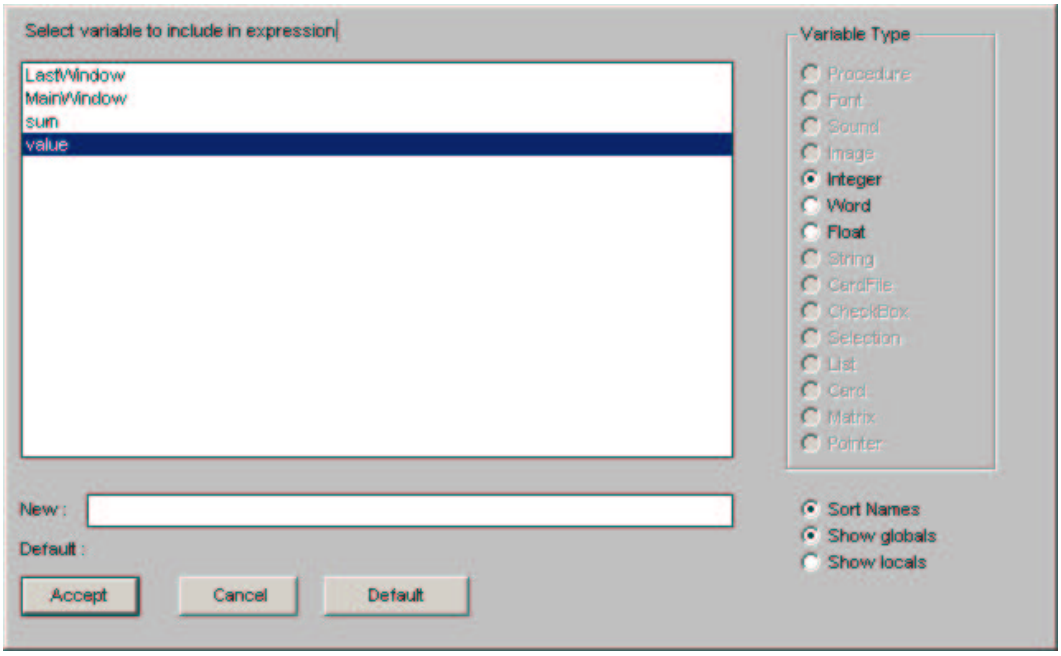

Figure 2: Selecting a variable.

the system are immediately encountered in this dialog. The program name is displayed in what appears to be a normal text-field, except that it is not editable. When the text-field is clicked to give it focus, a dialog appears having a regular text-field in it where the program name should be typed in. This annoying approach to enter values is used everywhere in the system.

To use this tool, the user must already know the basics of programming. The nodes that the user can place in the flowchart, and will have to in order to write any kind of useful program, include the fundamental building blocks of the Java and C languages, like assignments, control structures (`if`, `while`, `case`, `return`), procedures and so forth. Let us take this simple assignment expression as an example: `sum = sum + value`. A *DoAssign* node must first be created. The central node from Figure 1 is an example of a DoAssign node. The Large View of the DoAssign node is uninteresting, so it is not shown. It has two immutable fields, one for the left-hand side and one for the right-hand side of the assignment, that each bring up a dialog when clicked. The dialog for the left-hand side looks like the one in Figure 2 where the variable being assigned to is selected from a list of existing variables. Alternatively, a new variable can be created by typing its name into the text-field at the bottom. Figure 3 shows the dialog for editing the right-hand side of the assignment. An arbitrary expression can be typed in the provided text-field. However, the user cannot simply type the name of a variable into the text-field. They must press the *Select Variable* button to bring up a dialog like the one for selecting the variable on the left-hand side of the assignment. It looks like the tool does not like parsing user input; it expects the user to tell it explicitly that

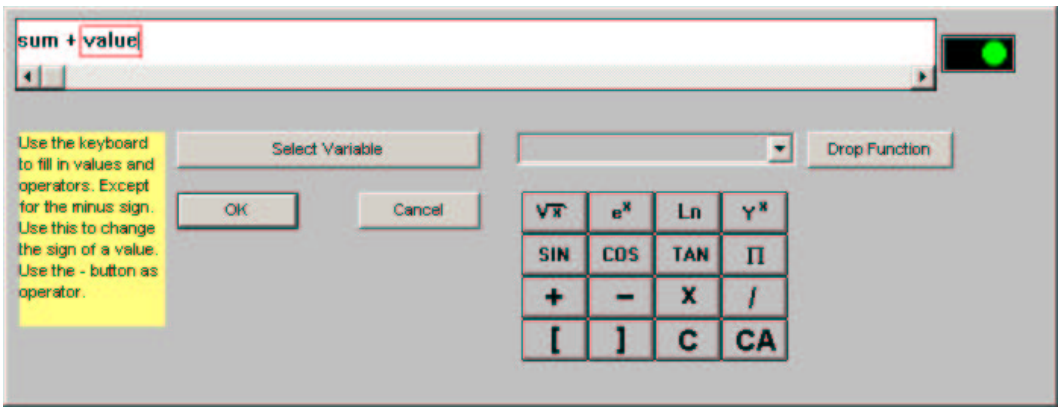

Figure 3: Editing the right-hand side of an assignment.

a variable is being used. There are buttons too for inserting basic arithmetic operations, square roots, powers, trigonometry functions and a few others. In the end, the user must know exactly what `sum = sum + value` does in order for them to understand what they have just done. In other words, it does not abstract out the basics of text-based programming in any way, while making it painfully tedious to accomplish anything.

The tool may be more useful for performing higher level tasks. The API supports GUI manipulation, loading and displaying images, some primitive image manipulation functionality, making database connections and more. These features were not explored in any detail because the tool tends to crash frequently. As a first impression, the API still seems quite tedious to use because the user is required to handle a lot of the low-level details, such as creating variables of the appropriate type. Perhaps an experienced user can learn to cope with these details, in which case Create may actually do something non-trivial. This tools was certainly the weakest tool of the ones reviewed. Subsequent tools showed more promise.

Some of the good points of this system are:

- Code can be generated in C or Java, so programs written with the tool are not bound to a particular language. It is not clear whether or not the tool can be extended to support other languages, or how sensitive it is to the particular system setup (which libraries and other tools that are installed).

Some of the downsides of this system are:

- Use of the tool is not at all intuitive, even for an experience programmer.
- The demo version of the tools crashes frequently. Saving, loading, and generating code always resulted in the tool crashing or hanging up. Hopefully, the commercial version of the tool is more stable.

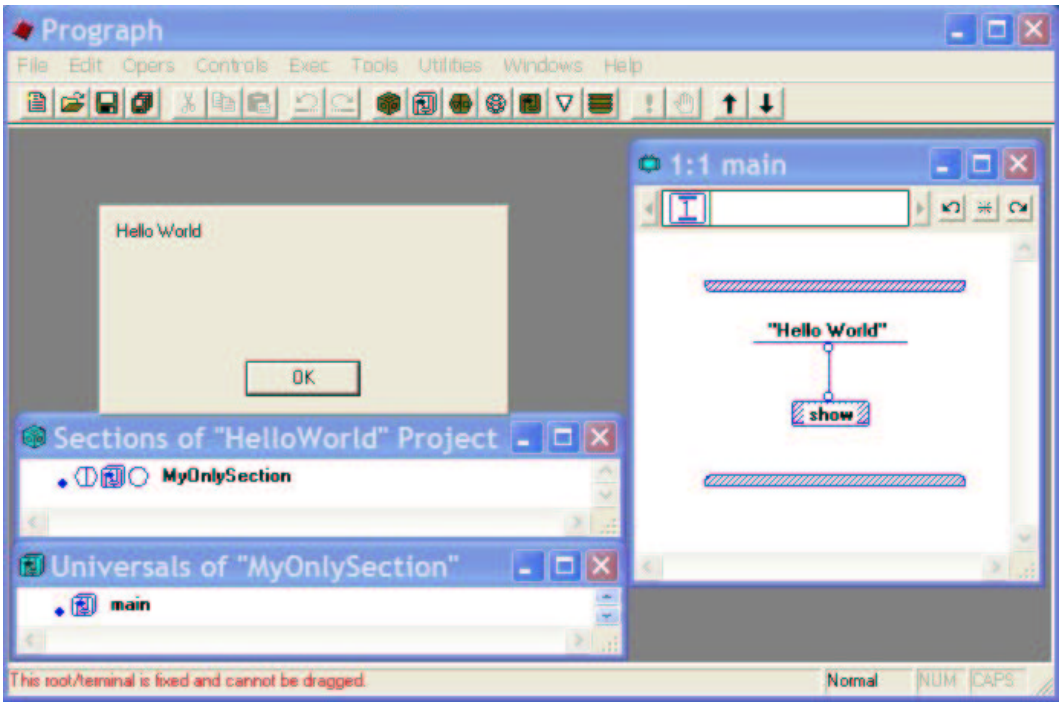

Figure 4: A *Hello World* program in Prograph.

3 Prograph

Prograph, by Pictorius [2], is a visual programming language, not a graphical front end to some other text-based languages. Where Create uses a control flow diagram to represent executable code, Prograph uses a data-flow diagram. Its programming model is a single-inheritance object-oriented one that allows stand-alone methods, called universal methods. Prograph also has the notion of persistents, which are like global variables whose states can be saved between executions of a program. The API consists of a set of built-in operations called primitives that behave much like universal methods. Primitives are divided up into several categories, the important ones being, bit operations, logical operations, file and I/O operations, list manipulation, math functions, string manipulation and memory management.

The user interface makes use of the Multiple Document Interface (MDI) model to display a program. MDI allows multiple child windows to be displayed within the confines of a parent window. Prograph uses MDI to supply a separate child window for editing the different elements that exist in a program, such as class declarations, method declarations and method bodies. It is therefore possible to edit several parts of a program concurrently.

To give the reader a taste of Prograph, Figure 4 shows the classic *Hello World* program. It consists of a single universal method called *main* in a section of the program called *MyOnlySection*. The rightmost window contains

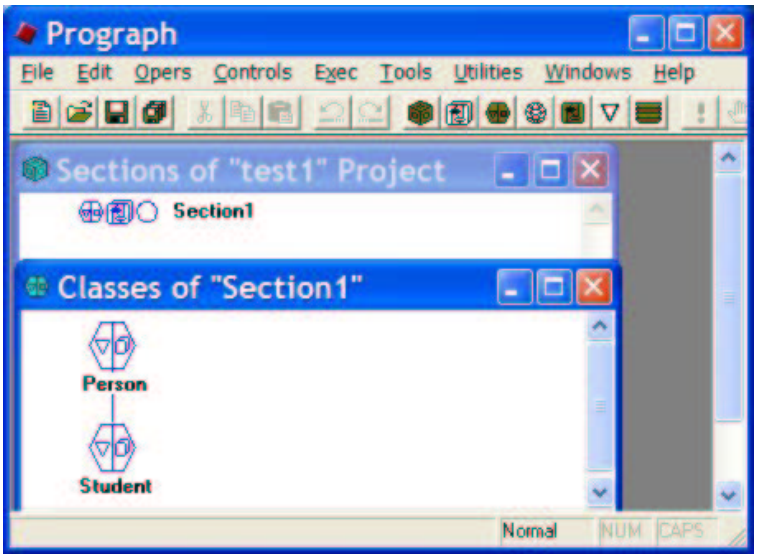

Figure 5: Prograph class declarations.

the body of the method, which consists of the string literal "Hello World" being passed to a primitive called *show*. At run-time, the show primitive displays a string in a window that has a single button used to dismiss the window. The window in the upper-left corner shows the program executing. While the entry point to this program happens to be a method called *main*, which is reminiscent of languages such as C and Java, execution can actually begin in any parameterless universal method.

Code for a program is divided up into sections, which are analogous to the source files of a program written in a text-based language. In fact sections literally are stored in individual files by Prograph. The sections of a program can be viewed by displaying the Sections Window, which can be done by selecting the *Sections* menu item from the *Windows* menu. The Sections Window shows a list of sections, where each section is displayed as three icons next to the section's name. The three icons that can be seen in Figure 4 represent the classes, universal methods and persistents of a section respectively. Creating a new section involves either double-clicking with the mouse on the background of the Sections Window, or selecting the *New Section* menu item from the window's context menu. Most elements of a Prograph program can be created in a similar manner. The section can be named when it is first saved to disk in a familiar way through a standard file chooser dialog.

Creating a new class is very easy. The first step is to display a section's Classes Window by double-clicking the hexagonal icon representing the classes of the desired section in the Sections Window. A new class is then created by double-clicking on the background of the Classes Window. An icon for the new class appears. By default the new class is called *Unnamed*, but a class can always be renamed. Clicking on a class's name replaces it with an

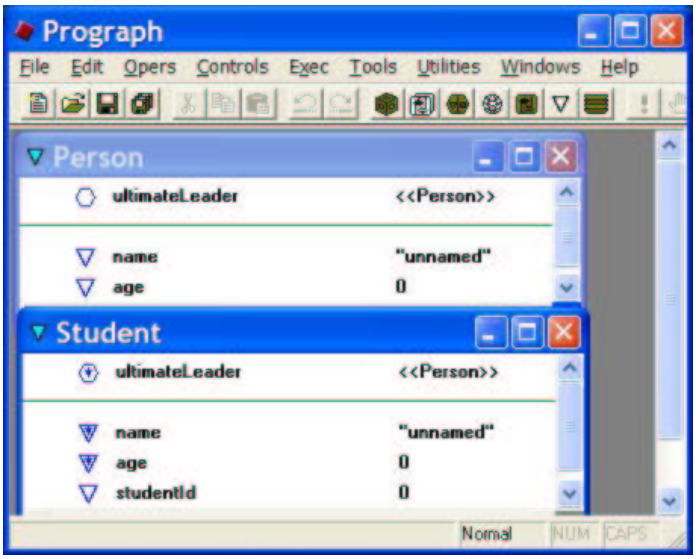

Figure 6: Editing class attributes.

editable text-field where a new name can be typed in. A class can be made the subclass of another by selecting the parent class and SHIFT-left-clicking on the subclass. A line extending from the bottom of the parent class's icon to the top of the subclass's icon is drawn to show the inheritance relationship. Figure 5 shows a class called Student that subclasses a class called Person. Note how the class icons are divided in two, with a triangular shape in the left-hand side and a rectangular shape in the right hand side. This feature will be relevant momentarily.

Classes can have two types of attributes: class attributes and instance attributes. These two types of attributes are analogous to static and instance variables in Java. To view or edit the attributes of a class, an Attributes Window like the ones shown in Figure 6 is opened by double-clicking on the left side of the class's icon. Class attributes appear above the horizontal line in the Attributes Window with a hexagonal icon, and instance attributes appear below the line with a triangular icon. All attributes show their default value in the rightmost column. Figure 6 shows that the Person class has one class attribute called *ultimateLeader*, and two instance attributes called *name* and *age*. The figure also shows that inherited attributes also appear in a class's Attributes Window. Inherited attributes can be distinguished from the other attributes by a little downward pointing arrow appearing within their icons. As expected, a new attribute is created by double-clicking in the appropriate region of the Attributes Window's background. As is the case with class names, the name of an attribute turns into an editable text-field when clicked. For simple types, the default value can be edited in the same manner; the type of the attribute will automatically be set accordingly. Editing the default value of an attribute whose type is a class is a little more complicated. The user can

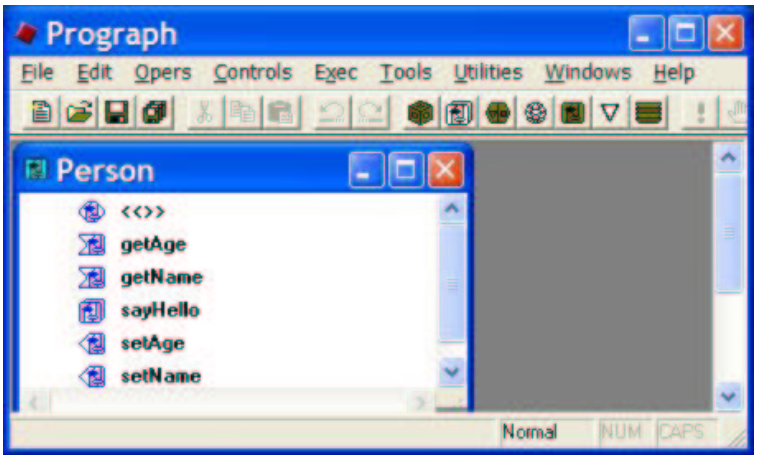

Figure 7: Editing class method declarations.

bring up a Value Window by double-clicking the attribute's icon. From this window, the user chooses the class of the attribute by selecting it from a list of existing classes. Because the attribute is of a class type, it too may have attributes whose default values may be set from the Value Window using this same approach recursively.

Editing the method declarations of a class is just as straightforward as editing its attributes. A Class Methods Window is first opened by clicking on the right side of the class's icon. A class can have four kinds of methods: Plain methods that can be used for performing any arbitrary computations, Get methods that return the value of an attribute, Set methods that set the value of an attribute, and Instance methods which are analogous to constructors in Java. The type of a selected method can be set through the menus. Figure 7 shows that each type of method has a different icon, and that instance methods have no name. It should also be noted from the figure that methods do not appear to have specific prototypes.

Creating universal methods is done in a similar manner, except that the Methods Window in which they are declared is opened by clicking on the methods icon of the section in which the methods belong in the Sections Window. Unlike class methods, universal methods are always of the Plain variety.

So far everything has been nearly trivial to do, and the graphical interface presents the relevant information well. For example, the user can clearly see the inheritance hierarchy as well as inherited attributes, which is not the case with text-based programming languages. However, aside from the *Hello World* program, we have yet to show any real executable code. Unfortunately, writing code is a considerably more complicated task.

Because code is represented using data-flow diagrams, there are no notions such as local variables and assignments. Figure 8 give a little bit more of an idea what the code looks like. Both methods create an object of type Person named *Bob* whose age is 2, then query the object for both its name and age

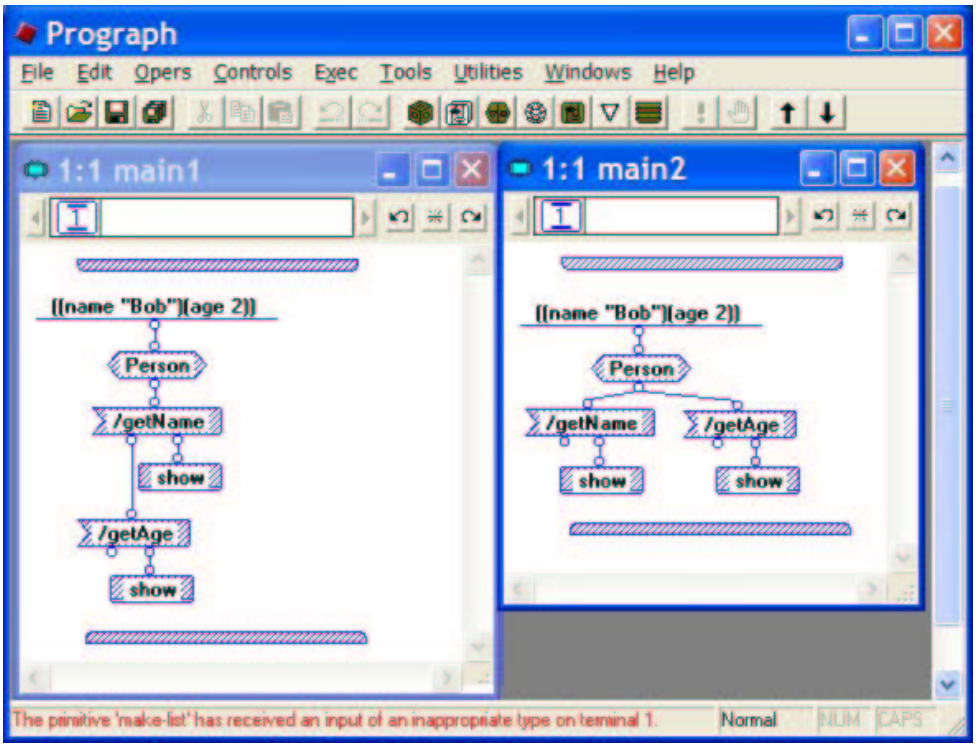

Figure 8: Creating and querying objects.

and displays those values. Note the cryptic notation used to initialize the Person objects. The topmost node in both methods is a list of lists containing key-value pairs, where the keys correspond to the names of the attributes of the object being initialized.

Each node in a diagram has input points, called terminals, at the top and output points, called roots, at the bottom. Nodes are connected together via data links. When a node executes, the data that it produces emerges from a root and travels over the data links connected to the root to the terminals at the other end. When input is available at all of a node's terminals, the node becomes eligible to execute. Normally, there is no guarantee with respect to the order in which nodes execute, so there is no telling whether *getName* or *getAge* will execute first in the second case of Figure 8. Most operations have a fixed number of roots and terminals. For example, Get operations, like *getName* which is a Get method, always have one terminal receiving the object from which an attribute is to be retrieved. Get operations always have two roots supplying the input object and the value of the requested attribute. Some primitives can have an arbitrary number of roots and/or terminals, such as arithmetic expressions and list packing/unpacking operations.

Method arguments arrive through roots on the shaded bar at the top of the method body and return values leave through terminals on the shaded bar at the bottom of the method body. So far, none of the examples have shown this

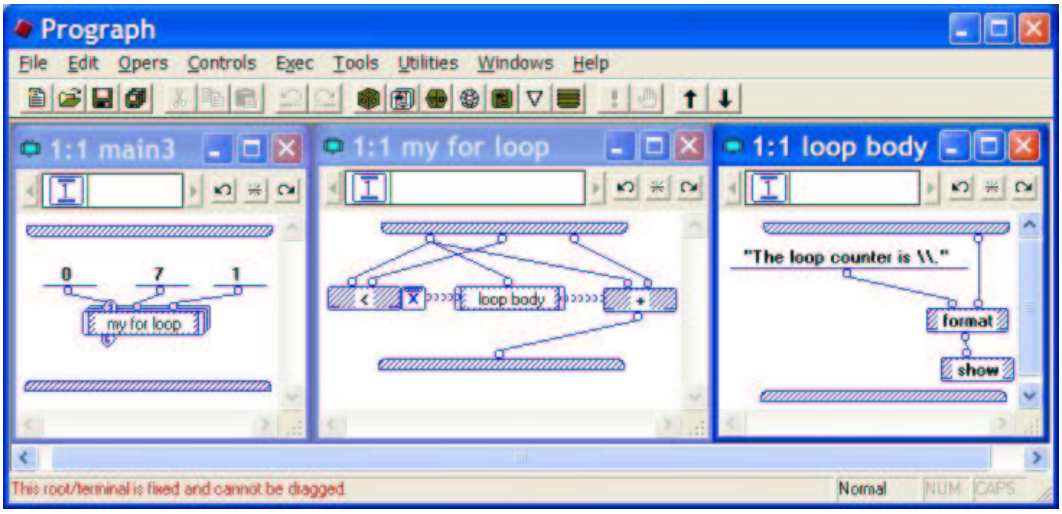

Figure 9: A basic `for` loop.

feature. New roots and terminals can be added by double-clicking near the edge of the bars. Doing so effectively modifies the prototype of the method.

Primitive and user defined operations can be transformed into list processing or looping constructs called multiplexes. For example, an annotation can be attached to a node's terminal via its context menu so that the node executes for every element of a list arriving at the terminal. This is a very useful feature because it completely abstracts out the iteration for the user. There is also an annotation that binds a root and a terminal belonging to the same node, so that the node executes repeatedly with the output value being fed back into the node as an input value. It is used to create loop constructs like ones typically found in procedural languages. However, implementing loops in this manner can be quite awkward. Figure 9 shows the implementation of a loop that is roughly equivalent to the following C code:

```
int i;
for (i = 0; i < 7; i++)
  printf("The loop counter is %d\n", i);
```

There are a couple of things to note about this loop. The first is that *my for loop* and *loop body* are not separate methods; they are local code blocks (pieces of code that behave as a single entity, but are edited in a separate window). The second is the presence of coil-like lines, called synchro links, between the three nodes in the *my for loop* block. Synchro links impose an ordering on the execution of the nodes that they are connected to. In this case they are used to ensure that the loop body executes after the loop condition is tested, but before the loop counter is incremented.

Some of the good points of this system are:

- The GUI is flexible: you can do pretty much everything through the menus, or using shortcuts (clicking on windows and icons).
- Providing section, class, attribute and method declarations is almost trivial.
- The user can have the interpreter execute any universal method with no inputs at any time. A universal method can therefore be used much like a Smalltalk workspace, which is good for debugging. If an execution error occurs, there is a Smalltalk-like debugger as well.
- There are search utilities for finding entities (classes, methods, etc.).
- Program elements that have been modified since the last time they have been saved are marked with a little diamond beside their icon.

Some of the downsides of this system are:

- Writing executable pieces of code is tedious, a problem made worse by the fact that linking nodes together is not always easy. The tiny circles have to be clicked exactly in order to create data links.
- It is too easy to create bogus entities by clicking on a window's background.
- Typos can be a problem. The user has to type in the text that appears in each node (class names, method names, primitives) and, depending on the node type, the user might not know about a mistake until run-time. Some types of nodes, such as arithmetic expressions, can be verified at edit-time. Context sensitive menus that contain the names that are already defined would be quite helpful.
- A link cannot be deleted by selecting it and pressing delete as one would expect; the user has to draw another link overtop of the existing one.
- The set of primitive operations does not appear to be extensible.
- The demo version tends to crash every now and then.
- The documentation is far from intuitive. One usually has to look in several different places in order to find all of the necessary information. For example, the part of the documentation that describes the purpose of data links and synchro links does not describe how to create them.

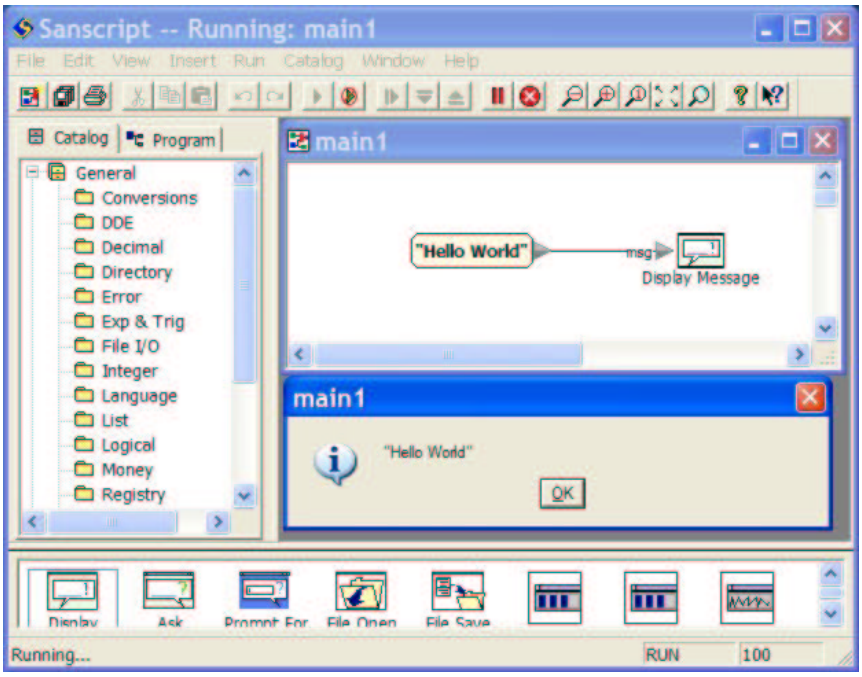

Figure 10: A *Hello World* program in Sanscript.

4 Sanscript

Sanscript, by Northwoods Software [3], is a tool that is very similar to Prograph, because it too represents programs using data-flow diagrams. However, Sanscript is not object-oriented, in fact, it is quite procedural. It has a lot of built-in types, including all of the usual ones, as well as files, errors, money and lists of these types. The available API is pretty extensive and contains quite a bit of high-level functionality for manipulating and formatting time, accessing the Windows registry keys, file and I/O operations, and accessing system resources.

Like Prograph, Sanscript's user interface also uses MDI for most of its editing purposes, however, it occasionally breaks the rule and opens an independent modal dialog. The interface also has a few other features not present in Prograph seen in Figure 10. On the left side of the screen is the Overview Window and at the bottom of the screen is the Catalogue Window. The Overview Window contains a listing of the different categories of functions available, including those defined by the user. When a category is selected in the Overview Window its contents are displayed in the Catalogue Window. To edit the definition of a function, the user double-clicks the function's icon in the Catalogue Window which causes its data-flow diagram to be opened in a child window. To make a call to a function, the user drags the icon from the Catalogue Window and drops it into the data-flow diagram where the call is to be made. This approach is somewhat more intuitive than Prograph's

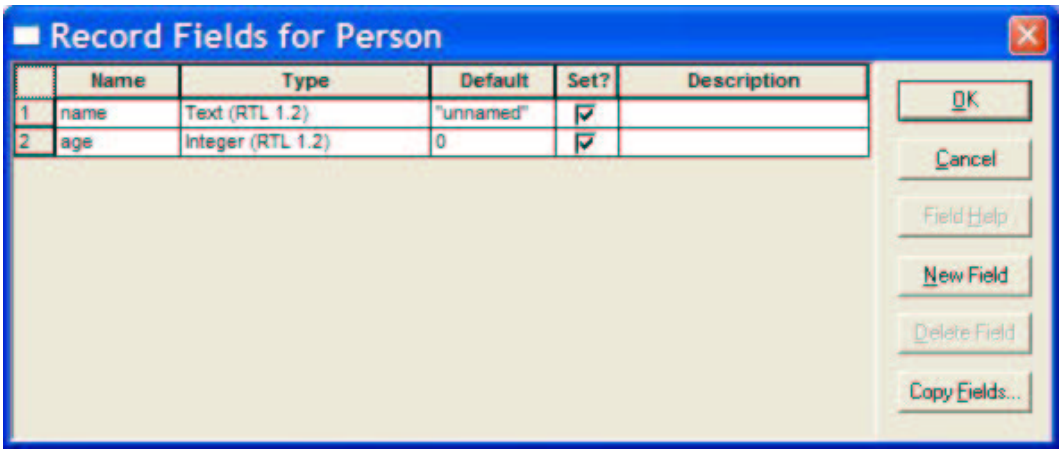

Figure 11: Editing the fields of a record.

approach of double-clicking in the background of a window or using a context menu to create a new node and typing in its name. Another subtle difference between Prograph's interface and Sanscript's is that Prograph's diagrams are laid out vertically whereas Sanscript's are laid out horizontally.

A quick note on terminology - input and output points on graph nodes are called inlets and outlets.

The user can create new types called records that are similar in nature to Pascal records or C structs. Doing so is not very difficult. The user first selects *File* -> *New* -> *Record* menu item. A dialog containing a text-field appears in which the user types a name for the new type. Then a dialog like the one shown in Figure 11 appears that allows the user to edit the fields of the record. Cells in the *Type* column are actually pull-down menus that list all available types. Default values can be typed directly into the cells of the *Default* column, however, the documentation does not describe if it is possible to specify default values for fields which are themselves records. There is no concept of a class (static) field as in Prograph, Java or C++.

Once a new record type is created, two new functions are created automatically and made available to the user. The first function creates and initializes new instances or sets the values of the fields of existing instances. The second function is used to retrieve the values of the fields. Figure 12 shows a program analogous to the Prograph program shown earlier that creates a Person object then displays the contents of its fields. It is far more concise than its Prograph counterpart for two reasons: the inputs do not have to be encoded as an obscure list of key-value pairs, and both fields can be accessed in the same operation.

Sanscript supports error handling, but in a slightly more primitive fashion than Java or C++ does. The user drops an error handler, which is a node much like any other except that it never has any inlets, into a function's body. If an error is generated within the body of that function or propagates up the

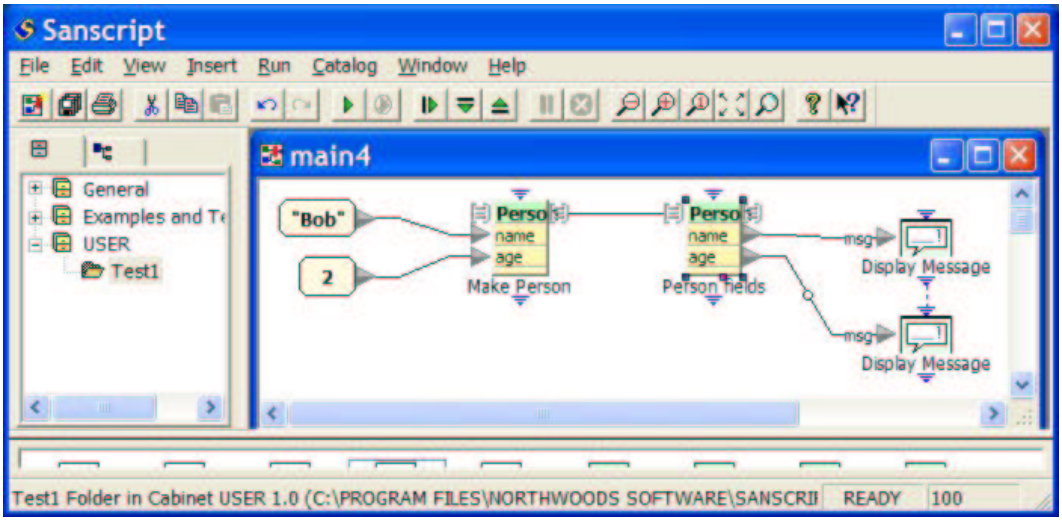

Figure 12: Creating a record instance and fetching its fields.

call stack because it was not handled from within a called function, the error handler executes. It is as though the entire body of the method were in a Java try-catch block that catches *Throwable*, the interface from which all of Java's exception classes inherit. The consequence of this approach is that the user cannot selectively choose the parts of the method that the error handler covers, nor which types of errors to catch. From within the error handler, the user can test which type of error was caught and handle the error accordingly.

Figure 13 shows a program called *main6* that calls a function called *Oops*. Oops unconditionally performs a division by zero which causes an error to occur. Though the body of the error handler seems quite large (see the bottom window in the screenshot), it does little more than extract some information from the error object and concatenate it into a string which is then displayed. Figure 14 shows the trivial outcome of running the program.

There are a couple of interesting control flow constructs. The first is a generalized case statement called *Pick One* that can operate on Booleans, integers, floating-point numbers or strings. In the case of Booleans, the construct boils down to an if-then-else statement. Making use of a Pick One node involves a three step process. First, the type of the operand must be identified. Then, the values for each case must be enumerated, unless the operand is a Boolean. Finally, a data-flow diagram must be supplied to handle each case. Figure 15 tries to demonstrate all three steps. The type of the operand is set automatically when an edge is connected to the node's inlet based on the type of data emerging from the outlet at the other end of the edge. In the upper-left window of the figure, a string literal is connected to the Pick One's inlet. Alternatively, the user can set the type manually by selecting the *Select Type* item from the inlet's context menu and picking the type from a list. Enumerating the values involves selecting the *Edit Choices* item from

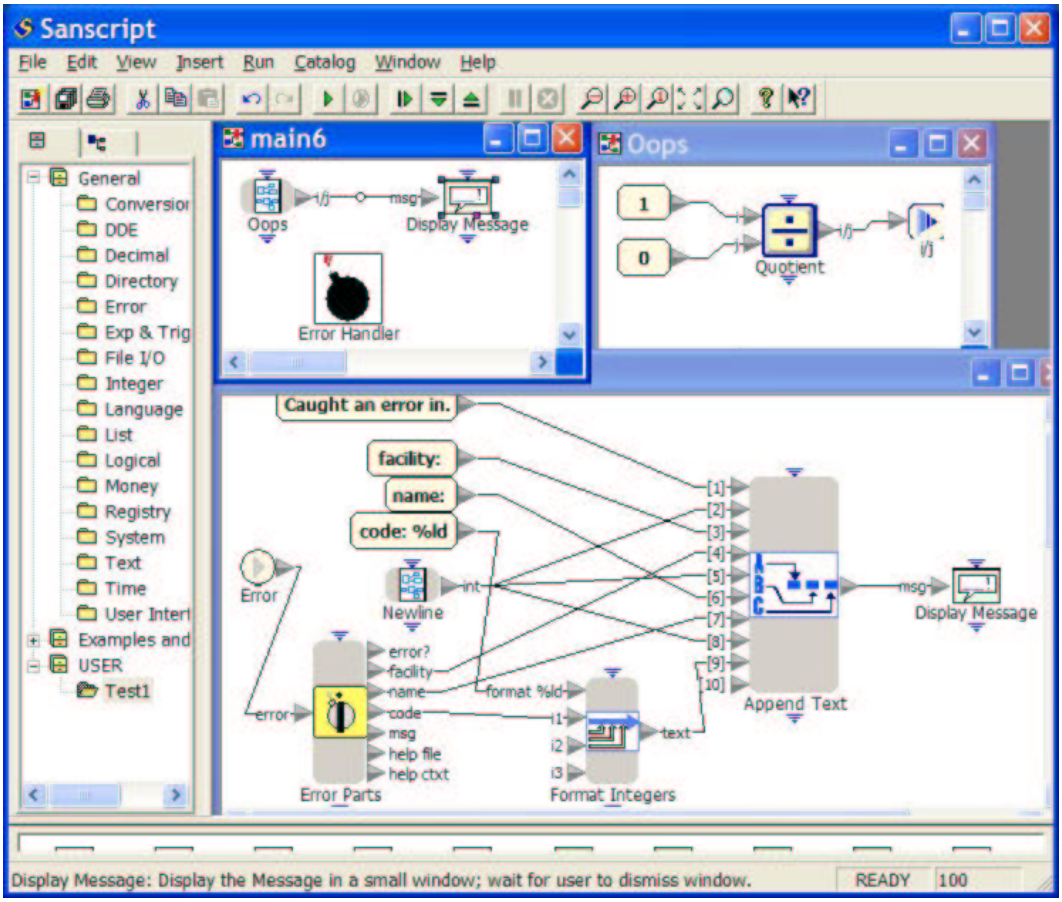

Figure 13: A program with an error handler.

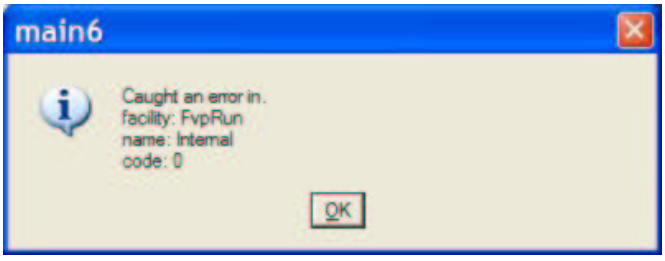

Figure 14: The output produced by the error handler.

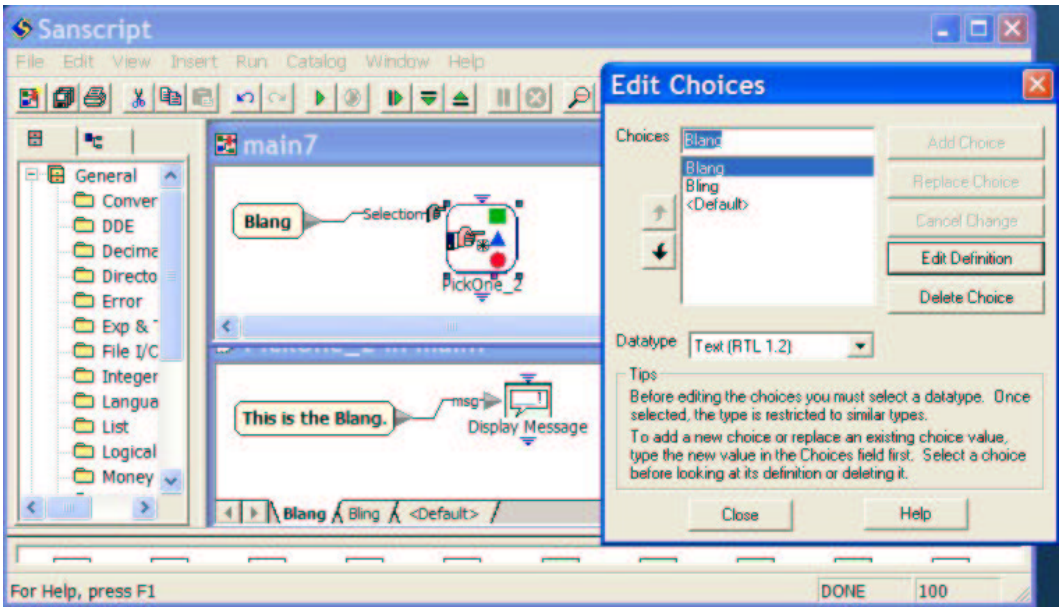

Figure 15: Configuring a Pick One graph node.

the node's context menu, which causes the dialog on the right to be opened. It has a text-field where the literal value for each case may be typed in, and buttons to manage the contents of the enumeration. The enumeration always contains an implicit *Default* member to handle arguments that do not match any of the other cases. This dialog is one example where editing is performed in an separate modal dialog instead of a child window. The bodies for each case are supplied by double-clicking the node itself which brings up a window like the one on the lower-left. The window has a tabbed-pane for each case, where each tab behaves like any other window used for editing diagrams.

The other interesting flow control construct is a *Repeat* loop, that can iterate over Booleans, integers or list elements. Like the Pick One, the type of the iterator is set appropriately when an edge is connected to the node's inlet, or it can be set manually by picking the type from a list. Given an integer the loop will iterate from 1 to the value of the integer inclusive. Figures 16 and 17 show two implementations of a loop that are similar to the C code presented in Section 3. The first iterates from 1-7 instead of of 0-6 as the C code does, but it was very simple to implement. It was only necessary to connect a 7 to a Repeat node to implement the iteration. The second is far more complicated and tries to remedy the situation by computing the value of the loop counter itself and breaking out of the loop at the appropriate time. Unfortunately, it has one minor flaw mentioned later. This implementation does not have an edge connected to the loop's usual inlet and no type is specified for the iterator, which effectively creates an infinite loop. The other inlets and outlets that have been added and the entire contents of the loop do practically the same thing as the Prograph implementation from Figure

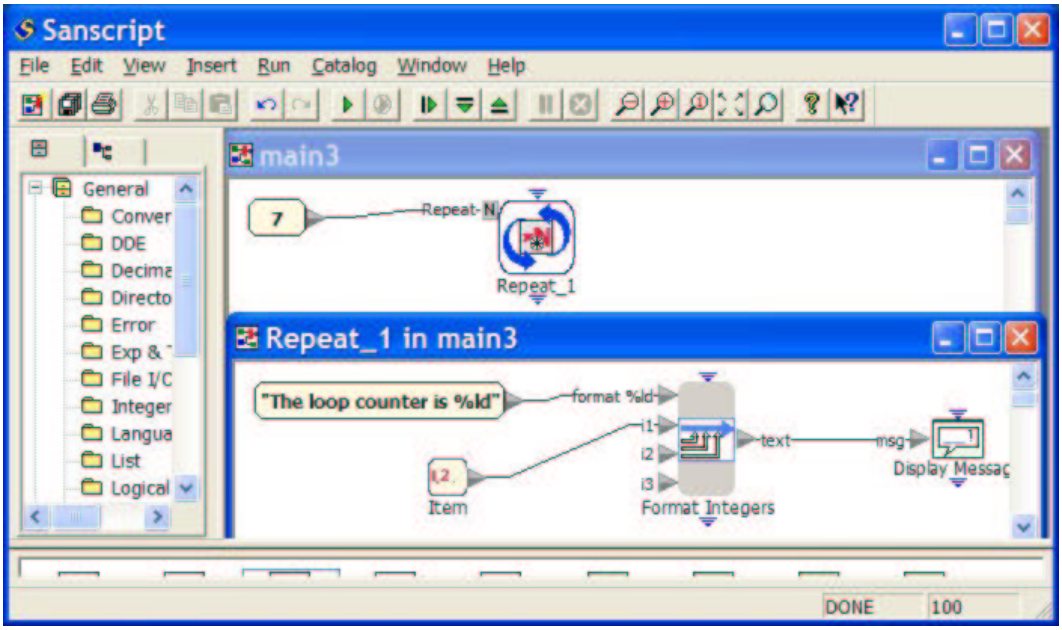

Figure 16: A simple `for` loop.

9. The only noticeable difference is that the loop body is inlined instead of being in a separate code block. It even has synchronization links to ensure the proper order of execution. However, there appears to be a semantic difference between Prograph and Sanscript. The Prograph loop breaks out immediately as soon as the loop condition fails, whereas the Sanscript loop breaks out only after the completion of the iteration in which the loop condition fails. So this loop actually iterates from 0-7 instead of the desired 0-6. To fix the loop, the loop condition would have to include a subtraction by 1.

Some of the good points of this system are:

- The system promotes code reuse. As in Smalltalk, all of the API that comes with the system and the user code is always available across projects. The API can be extended and shared; additional modules are available from the Northwoods Software web-site.
- The API is pretty extensive.
- Linking nodes together is easier than with Prograph because it tries to auto-complete edges based on where the user is dragging the mouse. On the other hand, this feature makes it a little too easy to create bogus edges if the user happens to click on an inlet/outlet. At least those edges are easy to delete; indeed, edges can be selected and deleted.
- The icons convey a lot more information than the Prograph icons do. Every icon is labeled and has pictures, instead of relying on obscure shapes.

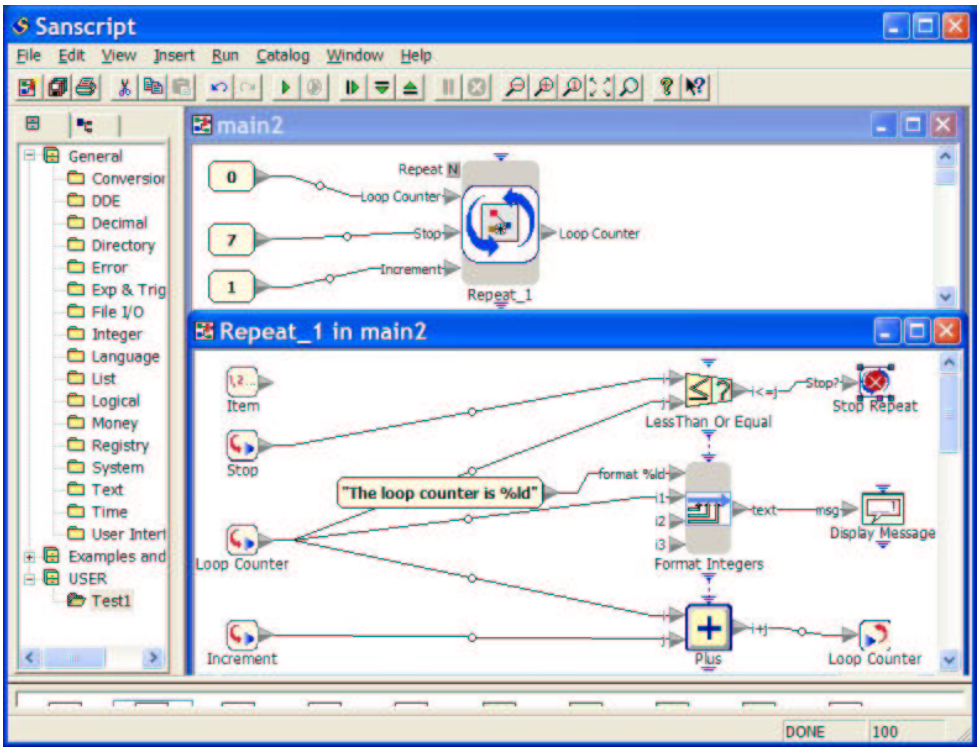

Figure 17: A `for` loop with customized iteration.

- It does not crash!

Some of the downsides of this system are:

- Once again, some simple things can be extremely tedious, such as basic text manipulation shown in the error handler example and more sophisticated control flow.

- When the user instantiates a node and edits its properties, most of the time the changes are local to that particular instance, as it should be, but in some cases the changes do propagate to all instances. For example, the functions to get/set fields of a record by default have outlets/inlets for each field. Inlets and outlets can be left unused, but sometimes it is desirable to remove unused ones to reduce visual clutter. So if an outlet is removed from one instance of the function, it is removed from all instances, usually resulting in disappearing edges. To avoid this problem, the user has to create a copy of the function, edit the properties of the new function, and use an instance of this new function instead.

- New record types are indirectly accessible/editable through instances of the functions that are automatically created for manipulating them, but do not seem to be available to the user otherwise. More importantly, there does not seem to be a way to delete them. For example,

if the user creates new types and deletes the functions associated with them, the types still exist in the system. This fact can be verified by the presence of the types in the pull-down menus wherever the user is asked to select a type. However, there no longer exists a way to create/manipulate/edit/delete instances of the type or the type itself.

- The icons are bulky (the size of the screenshots speak for themselves), but it is a trade-off from the extra information that is packed into them.

5 Starlogo

Starlogo [4], developed at the MIT Media Laboratory, is a specialized version of the Logo programming language that handles having multiple turtles carrying out actions concurrently. The turtles can act autonomously and sense each other as well as their environment. While it is well suited for teaching programming concepts to young students, it is intended to be used for simulating systems without centralized control (agents that act independently). Simulations are command driven and unfold visually within the user interface. At the lowest level, Starlogo is a text-based procedural language; all programming can be done in text. It also provides some higher level abstractions like GUI widgets for issuing commands and changing the values of variables. Moreover, code can be generated automatically from parameterized templates. The full version may be freely downloaded from the web-site.

The interface has two main windows: the *Control Center* and the other one is simply called the *Starlogo Window*. Both windows have the same main menu.

The Control Center, shown in Figure 18, is where the user writes textual code, much like they would in classic Logo. The window has two panes, one for turtle specific code and one for *Observer* code. The Observer is like a higher power than can create new turtles and has full control over the environment. Both panes are identical and are divided into two parts. The upper part allows the user to type in commands which are executed immediately. Turtle commands are executed by all turtles. The lower pane is where the user can write procedures that can later be executed as any other command. Nothing of a visual nature happens in this window.

The Starlogo Window has the animation area, controls for drawing and an area where users can create their own control widgets. Figure 19 shows the Starlogo window which has been shrunk down a whole bunch so that parts of it are missing. One of the neatest features of this window is that users can create their own buttons in the large white area and bind commands to them. The three blue buttons labeled *wander*, *grow-grass* and *setup* are examples of such buttons. Creating a button is accomplished by pressing the button creation button (the blue one with a finger on it in the tool-bar) and drawing a rectangle in the white area, much like one would in an image processing tool.

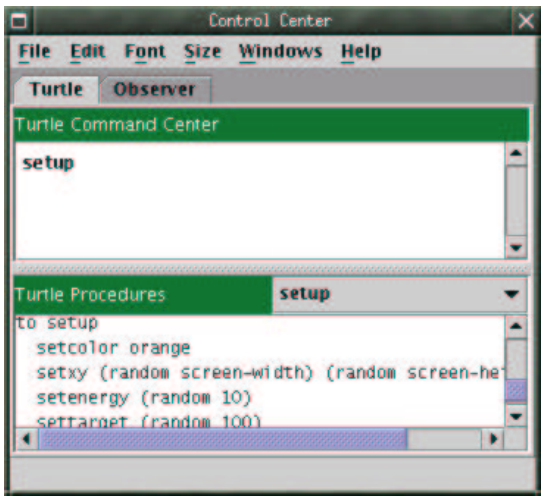

Figure 18: The Control Center.

A dialog like the one shown in Figure 20 is then displayed. In this dialog, the user types in a command that executes whenever the button is pressed and specifies whether the command is a turtle or observer command. The name text-field often does not render properly. If the *forever?* check-box is checked, the button fires repeatedly until it is depressed, instead of firing once when the button is pressed. Forever buttons are distinguishable from regular buttons by the chasing arrows appearing on them. Users can just as easily create sliders that allow them to change the value of a variable at run-time.

The system supports templates that can be used to automatically generate code. A template is used for describing an entire simulation, or the majority of it, not just one component of a program. Using a template, several breeds of turtles can be created that each exhibit different behaviour. The behaviour of a breed of turtle applies to all turtles of that breed, but it is possible for a turtle to change breeds during the course of its lifetime. This is analogous to an object that can change its calss at run-time. A template also allows the behaviour of the Observer to be specified. Since the code generated by a template clobbers all code in the Control Center, the interface only allows one template to be used per project. Templates are imported from external files, so it is possible for new templates to be added, but Starlogo does not come with tool support for creating new templates. In fact the documentation does not even address the issue of template creation. The current Starlogo distribution only comes with two templates. A screenshot of the Template Wizard for the *Ecology* template is shown in Figure 21.

The interface of the Template Wizard is divided into quadrants. The upper-left quadrant is used for creating and naming new breeds of turtles. Though it is not visible in the screenshot, the list always implicitly contains an entry for a default breed of turtle and the Observer, even though the Observer does not qualify as a breed of turtle. The lower-left quadrant shows the list of functions

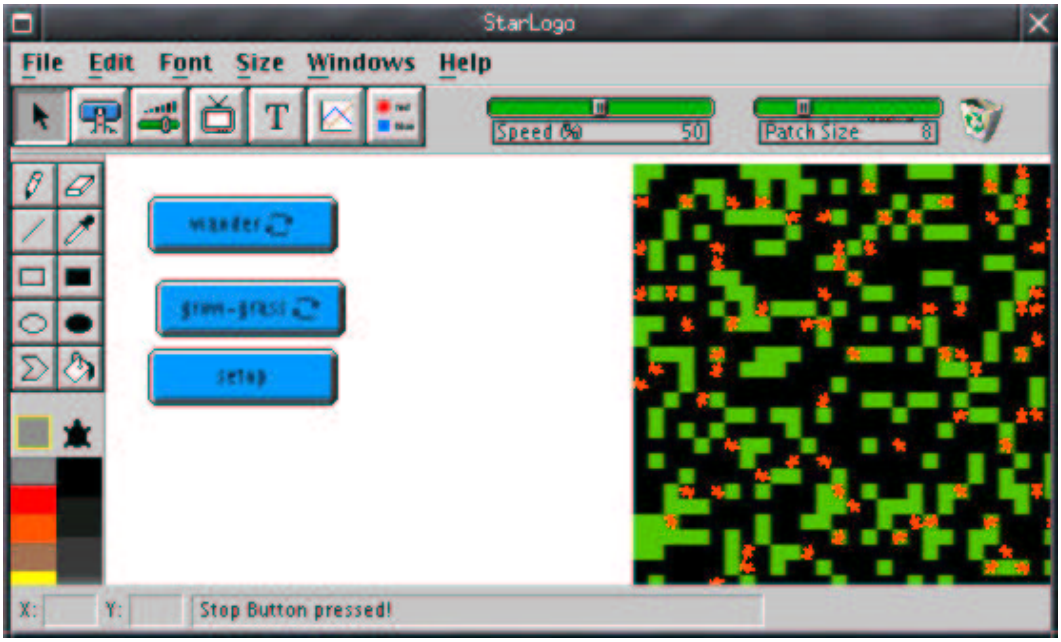

Figure 19: The Starlogo Window.

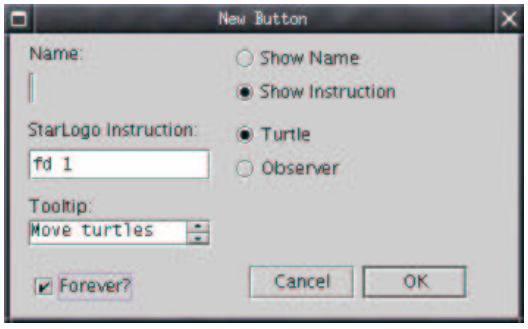

Figure 20: Configuring a button.

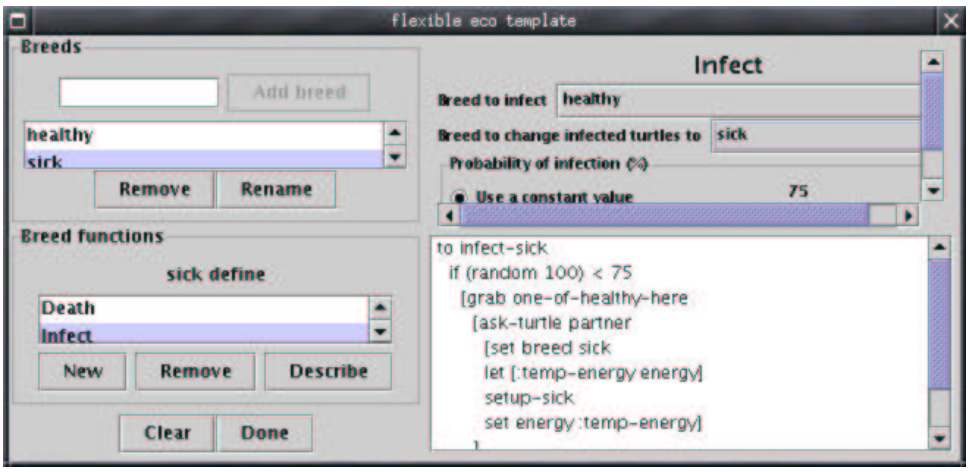

Figure 21: The Template Wizard.

describing the behaviour of the selected breed of turtle, or the Observer if it happens to be selected. The functions represent particular events that may occur during the lifetime of a turtle, such as walking, eating, reproducing, dying and so forth. The available functions may vary from one template to the other. It is also possible for the user to define new functions here, since it is not safe to write new functions in the Control Center. Recall that the code in the Control Control center is destroyed when the code for a template is regenerated. The upper-right quadrant shows the options that may be tuned for the selected function. Finally, the remaining pane displays the generated code for the selected function and is updated after every configuration change.

The configuration controls consist only of pull-down menus, radio-buttons and sliders that are are all accompanied by descriptive labels. Configuring a template is therefore, quite straightforward. On the other hand, it is unclear whether or not the interface is powerful enough to express complicated concepts, since the two available templates only request fairly simplistic information.

It only took a few minutes using the Ecology template to create a pretty complex simulation, with two breeds of turtles: sick and healthy. Sick turtles do not reproduce, they infect healthy turtles, they have less energy, they expend more energy to walk around, and they only partially consume patches of grass. Healthy turtles multiply randomly so long as they have a minimum amount of energy, eat all of a patch of grass, and have more energy initially. The rate at which the grass grows is also configurable. The result is fun to watch, as we can see the fluctuations in the overall population, the proportion of healthy versus sick turtles and the amount of grass. While that simulation was easy to create, there is no way to make a turtle die if it has been, say, infected by two different breeds of sick turtles. Adding that kind of functionality to the template would probably be difficult since there does not seem

to be a way to supply a list of conditions, much less making conjunctions or disjunctions of them.

Another interesting feature that Starlogo has is a Plot Wizard that allows the user to track data and build graphs as simulations unfold. The interface for doing so is rather large and spans a number of windows so it is impractical to include screenshots of this feature. First the type of plot must be chosen. The plot types are: line, bar, histogram, scatter and X-Y. Then the data to be plotted must be specified. Up to ten data items can be plotted at once. Specifying the data to be plotted is done through pull down menus, so it is not very complicated, but the interface is somewhat awkward. The problem with the interface is that the number of menus and their contents change dynamically depending on what is selected in them. To keep the discussion brief, the user basically has to specify the type of entity being observed (breed of turtle, patch of grass, etc.), the attribute being plotted (amount of energy, coordinates, age, etc.) and the way in which it is plotted (total, average, max, min, etc.). Experienced users can supplement the functionality available in the wizard by supplying their own plotting commands.

Some of the good points of this system are:

- No programming knowlege is required to generate code from a template.
- The tool is both text-based and visual.
- Allowing the user to add GUI widgets is practical.

Some of the downsides of this system are:

- Template code clobbers the contents of the Control Center. If the user wants to have their own functions they have to add them as custom functions through the Template Wizard.
- To configure user-created buttons and sliders, the user has to first CONTROL-click the widget to select it. Then it can be resized, or it can be double-clicked to bring up its properties dialog. Right-clicking the widget to bring up a context menu would have been more intuitive.
- The GUI is too dynamic. Widgets appear and disappear in response to user input.

6 ScriptEase

Neverwinter Nights, a fantasy role-playing game by BioWare, allows users to create their own modules depicting new worlds and scenarios. The game comes bundled with a toolset called Aurora that has several high-level features allowing the user to paint terrain, and populate the world with creatures, objects, triggers, special effects and more. Aurora also allows the user to create blueprints for creatures and objects that have new combinations of

properties. The actions and interactions that occur in modules are specified using a scripting language that resembles C. Writing script code with Aurora remains, for the most part, a manual task.

ScriptEase [5] is a pattern-based toolset for generating code in the Neverwinter Nights scripting language. Each pattern instance generates code for one small self-contained component of a user's module that acts independently from other pattern instances. Novice users can use ScriptEase without understanding the basics of programming, while those that do can extend the set of resources available to the toolset.

Because these tools make use of some concepts and terminology not normally found in programming literature, an introduction to the definition of a ScriptEase pattern is given in Section 6.1. Discussion of the use of the toolset follows in Section 6.2.

6.1 Pattern Definitions

At the top level, a pattern definition consists of two parts: a list of parameters and a list of situations. The role of a pattern parameter is similar to that of a function parameter; it becomes bound to a value that is used by an instance of the pattern, and has no significant impact on the structure of the generated code. Each situation describes a particular scenario to which a pattern responds, as well as how the pattern responds.

Let us use the *Icon Container* pattern to illustrate parameters and situations. The Icon Container pattern causes any number of actions to take place when an item, called the icon, is added to or removed from the inventory of an object, called the container. This pattern's two parameters are the icon and container objects. The three situations to which this pattern reacts are the cases when the icon is added to the container, the icon is removed from the container, and the icon is stolen from the container.

The definition of a situation, in turn, consists of four parts. Because user scripts can only execute in response to an event, the first part of a situation definition must necessarily be the specification of the triggering event. For the situation where the icon is removed from the container in the Icon Container pattern, the event of interest is the one where an item is removed from the container's inventory. Sometimes the occurrence of an event alone is not enough to determine whether or not a situation should react, so another part of the definition of a situation is the specification of a list of additional conditions that must also be satisfied. Ultimately, the conditions are joined together by conjunctions. Continuing with our Icon Container example, it is not sufficient for any item to be removed from the container's inventory; it is necessary for the removed item to be the icon. A situation definition also includes a list of actions that must be performed when the situation reacts (i.e. when the triggering event occurs and all of the conditions are met). The Icon Container pattern itself does not specify any actions; the responsibility is deferred to the user upon pattern instantiation, since the intent of the pattern is to allow any

number of arbitrary actions to occur. It is often necessary for the conditions and actions to manipulate values or objects other than the pattern parameters. For example, a plausible action may be to have a monster move toward the location of the container after the icon has been removed from it. To specify the action that moves the monster to the location, the location must first be defined. The definition of the location is an example of an entity. In addition to an event, conditions, and actions, a situation definition also contains a list of entities.

All elements of a situation definition are specified using atoms, where atoms consist of descriptive information and one function written in the Neverwinter Nights scripting language that performs exactly one task. Event Atoms, described below, are exceptional and carry two extra pieces of information used by ScriptEase when it binds script code to objects in a module. Atoms represent the smallest unit of computation available to the ScriptEase user. Because they are functions, atoms have parameters and may return a value. All atoms fall into one of four categories, where there is one-to-one correspondence between each category and the four parts of a situation definition. The categories are as follows:

Event Atoms specify a class of event and must, at the very least, have one parameter identifying an object to which script code is to be bound. These two pieces of information alone are almost sufficient for describing when a pattern situation should react. However, there are some specialized events which are described by the same class of event. For example, the removal and addition of an item from/to an object's inventory are both represented by an *OnDisturbed* event. When an OnDisturbed event occurs, a test may be performed to determine which specialization of the event actually occurred. The purpose of an Event Atom's function is to perform such a test, and return a Boolean value indicating whether or not an event which has occurred is the specialized event of interest. Most events do not have specializations. Event Atoms that represent events without specializations should always return true, since they have nothing to test. Because there exists a finite number of events that may occur in Neverwinter Nights, there is also a finite number of possible Event Atoms.

Entity Atoms always have a return value and are used for obtaining values or for extracting information from objects or the environment. An example was given earlier where a location had to be extracted from a container. Other examples include, obtaining a reference to the nearest door, and who the possessor of a particular item is. In short, Entity Atoms are used exclusively for obtaining data required within a situation, and should not have any side-effects.

Condition Atoms always evaluate to a Boolean value and are used for specifying the extra conditions that must be met before a situation reacts.

Action Atoms define actions as their name suggests, where an action can basically be any code having some sort of side-effect, such as having a monster attack a player character. Action Atoms may also optionally have a return value if they create a new object, or compute a value that may be needed subsequently. In that respect, they also behave like Entity Atoms, however, they may not be used where Entity Atoms are expected.

The type of a parameter, pattern or atom, and the type of an atom's return value may be one of a handful provided by ScriptEase. The set of available types includes the entire set of types native to the Neverwinter Nights scripting language, which in turn includes primitive types such as `integer`, `float` and `string`, as well as other higher-level types such as `location` and `object`. The remaining types consist of refinements of the native types that come in two forms: enumerations of integers, and aliases to other native types. These refined types exist in order to restrict the values that may be assigned to a parameter to those which are appropriate given the parameter's context, and to convey more precise information to the ScriptEase user. The enumerated types are used for describing things such as Booleans, spells and effects that do not have native types in the scripting language, but that are instead simply expressed by symbols predefined by the compiler that map to integral constants. As for the aliased types, some examples include `Item` and `Door`, which are both aliases to the `object` type, a type used to express every game object spanning creatures, player characters, items and so forth. Unlike the set of enumerated types, the set of aliased types is not extensible, since there is no real need to do so.

6.2 Toolset

ScriptEase consists of three visual tools appropriately named the Atom Builder, the Pattern Builder and the Instance Builder that supplement, but do not replace, BioWare's Aurora toolset for building modules. The *Atom Builder* is used to create and edit atoms, as well as enumerations. Of the three tools, the Atom Builder is the only one that requires knowledge of the Neverwinter Nights scripting language on behalf of the user. The *Pattern Builder* is used to create and edit patterns. Finally, the *Instance Builder* is used to create and adapt pattern instances for a particular Neverwinter Nights module developed using the Aurora toolset. From the pattern instances, the Instance Builder also generates script code for them and updates the contents of the module file accordingly.

The main window of all three tools are nearly identical, so the Atom Builder is used as an example. It can be seen on left hand side of Figure 22. The main window only provides a report of the number of things that are currently defined, and a menu-bar. The *Edit* menu contains one menu item for each category listed in the main window. Selecting one of these menu items displays

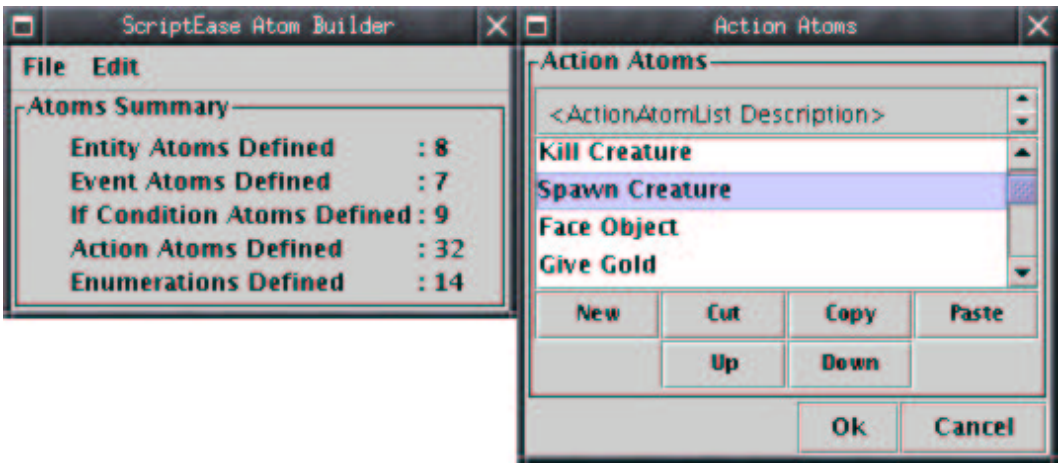

Figure 22: The Atombuilder's main window and the list of Action Atoms.

a dialog like the one on the right hand side of Figure 22, that contains a list of all items in the selected category. In this example the list of Action Atoms is displayed. Lists like this one occur frequently in all three tools to represent lists of atoms, patterns, pattern instances, situations and parameters. A new element can be added to the list by pressing the *New* button, and an existing list element can be edited by double-clicking it. In both cases, a new dialog is displayed for editing the list element.

Let us take a closer look at the individual tools, starting with the Atom Builder. Figure 23 shows the dialog for editing an Action Atom and how the return type[1] is selected by choosing it from a pull-down list of available types. Note how the parameters appear in a familiar looking list. The dialog for editing a parameter (not shown) has a pull-down list like the one for selecting the return type, and two text-fields: one for supplying the parameter's name and one for supplying a brief description. The code for the atom is supplied via a built-in editor that is displayed when the *Edit Code* button is pressed. The editor is rather featureless; it is just a general-purpose text-area where the body of the atom's function is typed in. The function's prototype is not typed into the editor, because it is generated from the information supplied in the atom's main dialog. Otherwise, the user supplies a name for the atom, and some descriptive information in text-fields. For event atoms, the user must additionaly pick one of the parameters from the list. It is through this parameter that a situation using the Event Atom passes the object to which the situation's script should be attached.

Creating an enumerated type involves a very short process depicted in Figure 24. The user gives a name to the enumeration and supplies a list of key-value pairs in text-fields, where the keys are descriptive names and the values may either be literal integers or symbolic constants predefined by the

[1]Only Entity and Action Atoms have configurable return types.

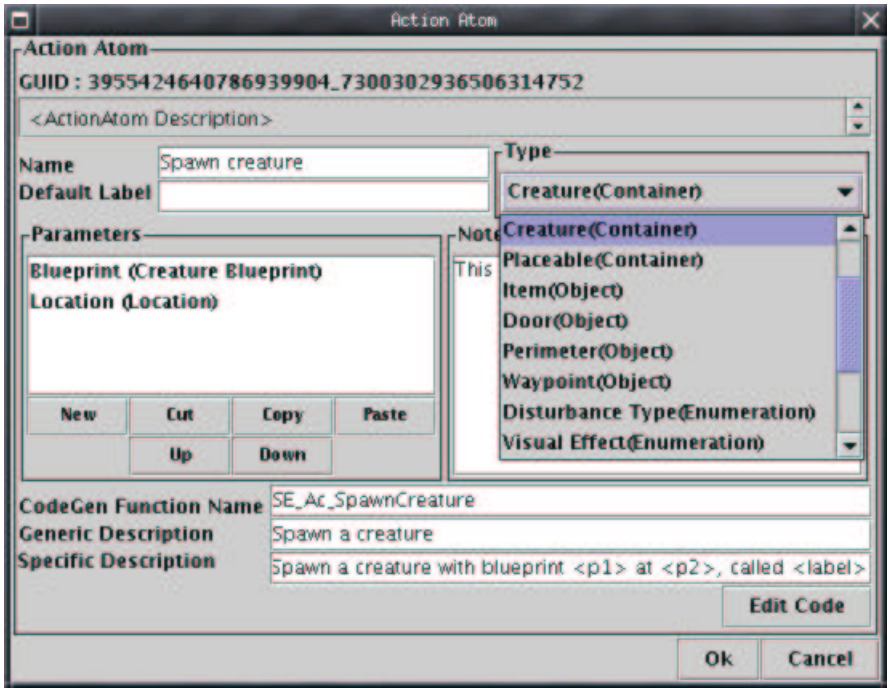

Figure 23: Selecting the return type of an Action Atom.

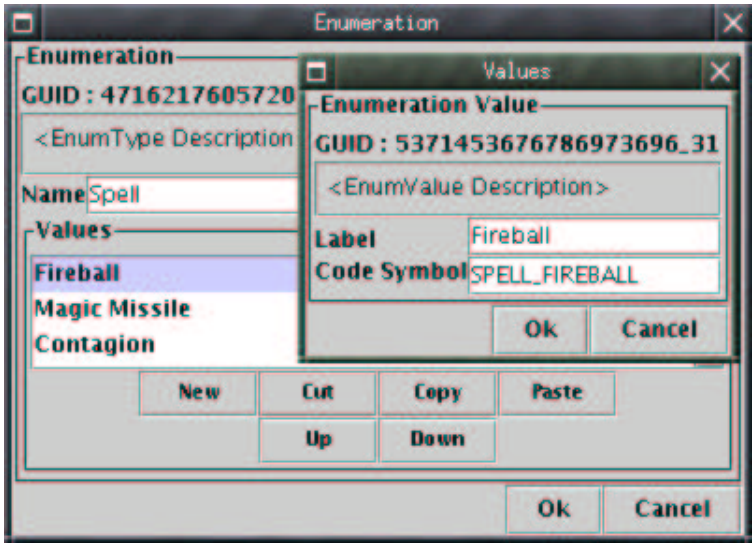

Figure 24: Adding an element to an Enumeration.

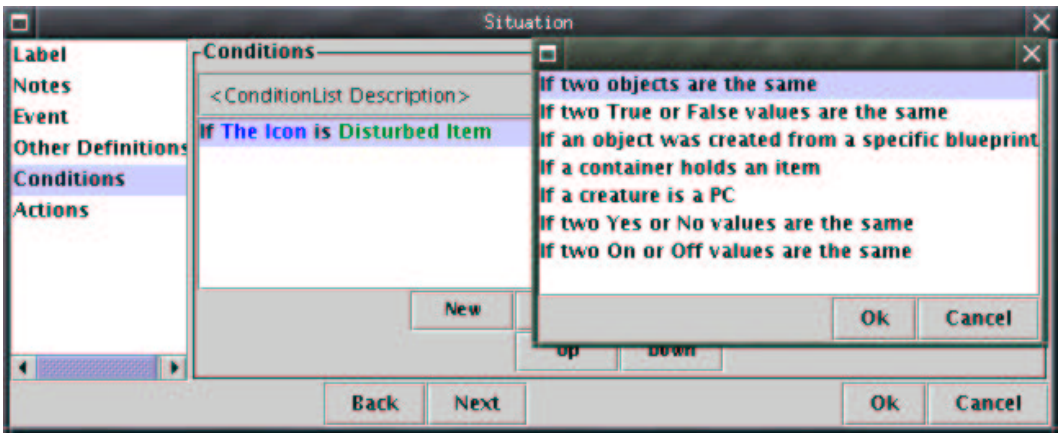

Figure 25: Adding a condition to a situation.

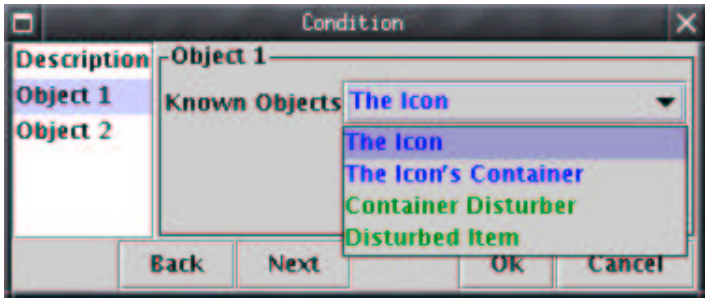

Figure 26: Binding an atom's parameters.

compiler.

To create a new pattern with the Pattern Builder, the user first supplies a list of parameters in the same way that atom parameters are supplied in the Atom Builder. The user then creates new situations and edits them in the Situation Editor. In the Situation Editor, the user picks the triggering event from the list of Event Atoms, and adds an arbitrary number of additional definitions, conditions and actions by picking atoms from the list of Entity, Condition and Action Atoms respectively. Figure 25 shows the process of adding the condition that the Icon must be the disturbed item to the situation where the Icon Container pattern by selecting the Condition Atom that compares two objects. For every atom added to a situation, the user is presented with a dialog that allows them to bind the atom's parameters. An atom's parameter may be bound to any of the pattern's parameters or any other entity of the appropriate type defined by another Entity or Action Atom. Figure 26 shows the first object of the object comparison atom being bound to the pattern's icon parameter. The pull-down list of valid bindings distinguishes between pattern parameters and other entities by displaying them in different colours.

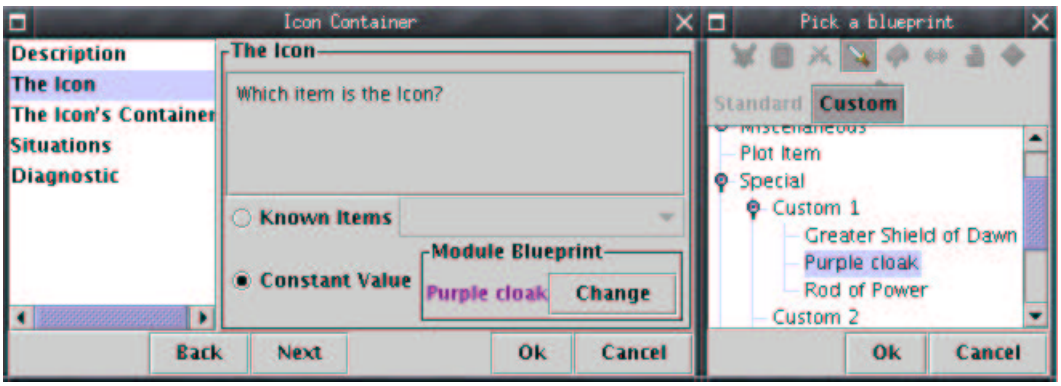

Figure 27: Editing a pattern instance.

Before instantiating patterns with the Instance Builder, a module containing terrain, creatures, and objects must first be created using the Aurora toolset in the usual manner, with the only exception being that scripting tasks may be omitted. The module may then be loaded into the Instance Builder and pattern instances added to it. Given a rich enough set of patterns and atoms, the Instance Builder is the only one of the three tools that ScriptEase users actually have to use. It is therefore possible for novices to avoid writing any code at all for a module.

To add a pattern instance to a module, the user selects the desired pattern from a list. Of course, it is solely the user's responsibility to decide which pattern is best suited for the task at hand. A dialog like the one from Figure 27 is displayed through which the user adapts the pattern to the needs of the application in two ways. The first form of pattern adaptation, which is not optional, is the binding of values to each of the pattern parameters. Depending on the type of parameter, the user either supplies a literal value for primitive types, chooses a value from a list for enumerated types, or chooses a value defined in the game module for the higher-level types. Figure 27 shows the latter approach for a parameter of type Item. Pressing the *Change* button, displays a dialog containing a tree listing all of the items from the game module. The second form of pattern adaptation, which is an optional one, is accomplished by editing the definition of the situations. A pattern instance is actually a prototype of the pattern from which it originated, meaning that the definition of the pattern is copied into the instance, where it may vary without affecting the original definition or other instances. When editing the definition of a situation within a pattern instance, the user is presented with a Situation Editor that is nearly identical to that of the Pattern Builder tool, with one subtle difference; when binding atom parameters, the values defined within the module are also made available. Normally, the existing definitions in a situation should not require modification upon instantiation of a pattern, otherwise, the design of the pattern would be questionable. However, it is common practice to add more definitions to a situation, such as actions in an

instance of the Icon Container pattern, which would otherwise do nothing.

Enumerated types, atoms, patterns and pattern instances built using the toolset may be exported to files in XML format. This feature is needed so that the output of one tool may be used by another tool in the toolset. For example an atom built using the Atom Builder may then be used in a pattern created with the Pattern Builder. It also allows users to easily share resources. As a precaution, the tools check on which atoms patterns and pattern instances depend before exporting or importing them, to ensure that pattern definitions remain intact.

Some of the good points of this system are:

- No programming knowledge is required when using available resources (patterns and atoms).
- ScriptEase's ability to load values defined in a game module and make them available to the user when binding parameters helps to reduce errors and relieve the burden of recalling details.
- The controls are straightforward and easy to use. Most activities involve picking something from lists, pressing buttons, or typing short pieces of text into text-fields.
- Fairly complicated tasks can be expressed with little effort if all of the necessary resources exist already.

Some of the downsides of this system are:

- ScriptEase is not as powerful as the Neverwinter Nights scripting language alone.
- The main window of all three tools does not provide any useful functionality.
- There are too many modal dialogs. Anything that can be edited opens up in a modal dialog making navigation extremely difficult. It is easy to forget which dialog is active, and the user has to close what he/she is doing to look up some information somewhere else. Things get even more confusing if all three tools are open simultaneously, because their windows rapidly proliferate and intermingle.
- Double-clicking list items to edit them is somewhat counter-intuitive. To edit something, we usually expect to select it and press a button or select an option from the main menu or a context menu.

7 Lessons Learned

The very nature of visual programming is quite high-level, therefore it is best suited for dealing with high-level concepts. It is hard to provide useful high-level abstractions for low-level concepts. Consider the assignment of a simple expression to a variable (e.g. `sum = sum + value;`). No matter how this concept is represented, the assignment, its left-hand side and its right-hand side have to be expressed in as much detail. Representing it visually does not provide a simpler, more elegant, or easier to understand abstraction than a textual representation. The learning curve also remains the same, not to mention that people are actually more accustomed to seeing such simple constructs in textual form; it resembles a math equation from an elementary school textbook. So tools like Create that provide a near one-to-one mapping of a procedural text-based language to a visual graph make for a poor application of visual programming. On the other hand, Starlogo's Template Wizard and ScriptEase's Instance Builder only manipulate high-level abstractions and are easy to use for their intended purposes.

Though it may be claimed by some that visual programming is more natural for humans that programming through the interim of text-based languages, we have to consider the relative ease with which humans can perform the programming task. Most programmers are adept at typing and can do so rapidly regardless of what they are trying to express. Building graphs with the common tools for interacting with a computer – a mouse and a keyboard – is usually considerably more tedious and time consuming than typing text. Visual programming also requires learning how to manipulate and use a wide variety of different visual elements, whether they be graph nodes or dialog boxes.

Visual programming tools should provide as much support as possible for frequent tasks, otherwise novice programmers will have a hard-time learning and experienced programmers will rapidly become annoyed. For example string manipulations are very bulky and awkward in Sanscript. It does not even supply an abstraction for a newline character; a character with value 10 (the ASCII equivalent) has to be constructed. Likewise, both Prograph and Sanscript have different forms of high-level support for loops, like looping over list elements, but doing other forms of loop indexing quickly becomes painful.

Humans usually do not work in a linear fashion; we tend to jump around from one task to another, or work on several subtasks concurrently to create a greater whole. Navigating a project and accessing functionality should therefore be quick and easy in a visual programming tool. Both Prograph and Sanscript have a main window that remains fairly static as the user works (it is never replaced with another window), and allow multiple views to be opened simultaneously. Most of the functionality is available through the main window's menus and other controls, so there is usually just one or two degrees of separation between what the user is doing at any given moment and the functionality and/or views that the user may want to access the next.

On the same token, modal dialogs should be avoided whenever possible because as long as one is visible, it will prevent all navigation. This prevents users from multitasking, or simply from looking up information that is needed to deal with the dialog. The last point is a particularly brutal blow to novice users that want to see how they have dealt with other instances of the same dialog before. ScriptEase and the Aurora toolset as well are particularly bad with respect to modal dialogs.

Tools should not hide details known to the user, like records in Sanscript. It confuses and frustrates users if they cannot find something that they wants to recall or change.

Finally, consistency is of the utmost importance, whether it be how similar tasks must be carried out by the user, or how data is represented in the GUI. Consistency helps to shrink the learning curve, and reduce mistakes and confusion.

8 Conclusion

This paper surveyed five visual programming tools: Create, Prograph, Sanscript, Starlogo and ScriptEase. Create, Prograph and Sanscript use diagrams to represent code and are as expressive as text-based programming languages. While they can be used for writing arbitrary programs and provide some high-level constructs, they are not very practical to use and often fail to simplify basic programming tasks with respect to their text-based counterparts. Starlogo and ScriptEase use higher-level abstractions – templates and patterns – to automatically generate textual code. Their use is straightforward and easy to learn, even though their interfaces require some refinement. They can generate elaborate pieces of customized code with little effort on behalf of the user. However, they do not have the expressive power that their underlying text-based languages possess. Additional lessons learned from the survey are that visual tools are well suited for manipulating high-level abstractions, should provide a lot of support for frequent tasks, and promote ease of navigation and consistency.

References

[1] Sharper Software. Create. http://www.create-software.com.

[2] Pictorius Incorporated. Prograph. http://www.pictorius.com/prograph.

[3] Northwoods Software. Visual programming tools from Northwoods Software. http://www.nwoods.com/sanscript.

[4] MIT Media Laboratory. Starlogo on the web. http://education.mit.edu/starlogo.

[5] M. McNaughton, J. Redford, J. Schaeffer, and D. Szafron. Pattern-based AI scripting using ScriptEase. In *AI'2003: The Sixteenth Canadian Conference on Artificial Intelligence*, pages 35–49, June 2003.