VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui

Department of Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 471 5328
Fax: +64 4 495 5232
Internet: Tech.Reports@comp.vuw.ac.nz

Object Oriented Program Visualisation in Tarraingím

James Noble, Lindsay Groves, and Robert Biddle

Technical Report CS-TR-95/15
September 1995

Abstract

Most program visualisation systems display either language level details of programs, or high level overviews of the program's algorithm. Producing high level views requires that the abstractions in the program to be visualised (the *target program*) are described to the visualisation system — either by modifying the target program, or by providing the visualisation system with information about the details of the abstractions in the target program. Object Orientation can be used to organise the design of both the target program and the architecture of an abstract program visualisation system. Such a system can produce flexible displays of the target program at multiple levels of abstraction, without any extra detailed information about the implementation of abstractions in the target program.

Publishing Information

This was originally published as: James Noble, Lindsay Groves, and Robert Biddle, "Object Oriented Program Visualisation in Tarraingím", *Australian Computing Journal*, 27:4, November 1995.

Object Oriented Program Visualisation in Tarraingím

James Noble, Lindsay Groves, and Robert Biddle

Abstract

Most program visualisation systems display either language level details of programs, or high level overviews of the program's algorithm. Producing high level views requires that the abstractions in the program to be visualised (the *target program*) are described to the visualisation system — either by modifying the target program, or by providing the visualisation system with information about the details of the abstractions in the target program. Object Orientation can be used to organise the design of both the target program and the architecture of an abstract program visualisation system. Such a system can produce flexible displays of the target program at multiple levels of abstraction, without any extra detailed information about the implementation of abstractions in the target program.

1 Introduction

Program Visualisation is the application of computer graphics techniques to computer programs, in the same way that scientific or engineering visualisation applies these techniques to scientific data or engineering artifacts. Visualisations can assist programmers in constructing, debugging and maintaining programs. They can also be used in teaching general principles of computer science, including the design of data structures and algorithms.

This paper describes our experience in using object orientation to organise a program visualisation system. In particular, we describe the design of a program visualisation system called Tarraingím (from the Gælic *to draw*), which visualises programs written in (and is itself written in) the object oriented programming language, SELF (Ungar and Smith, 1987). Because Tarraingím understands the basic structure of the language, it can display language level views of objects, produce execution traces, and generate program execution profiles. Tarraingím also allows the programmer to define higher level views of objects, and by monitoring the program, can keep these views correct as the program executes.

Section 2 presents the idea of *abstract program visualisation* — that a visualisation system should illustrate a program in terms of the abstractions comprising the program's design. Section 3 describes how object orientation can be used to organise the target program so that all its abstractions can be visualised. Section 4 outlines the design of Tarraingím, which is constructed as an object oriented framework, and Section 5 presents some examples of the kinds of visualisations that Tarraingím currently produces. Section 6 compares Tarraingím with recent related work. Section 7 discusses experience with Tarraingím and presents some conclusions.

2 Abstract Visualisation

The existing work in program visualisation can be grouped into two broad categories, according to the kinds of views presented. *Algorithm Animation* systems (Brown, 1988a; Stasko, 1990; Bentley and Kernighan, 1991) display high level pictures of the operation of an algorithm, while *Graphical Debuggers* (Myers, 1983; Shilling and Stasko, 1994) display language level views of a program.

Algorithm animation systems illustrate the essence of an algorithm. The displays they produce may bear no direct relation to that algorithm's implementation, and so must be specially designed. Typically, the algorithm to be visualised is implemented using facilities provided by the visualisation system, so that as it gruns the animated displays are updated.

Graphical debuggers (and graphical programming environments) display information about a program's implementation. A graphical debugger illustrates run-time control flow and data structure, while a programming environment presents static views which may be edited to create programs. For example, a debugger may display

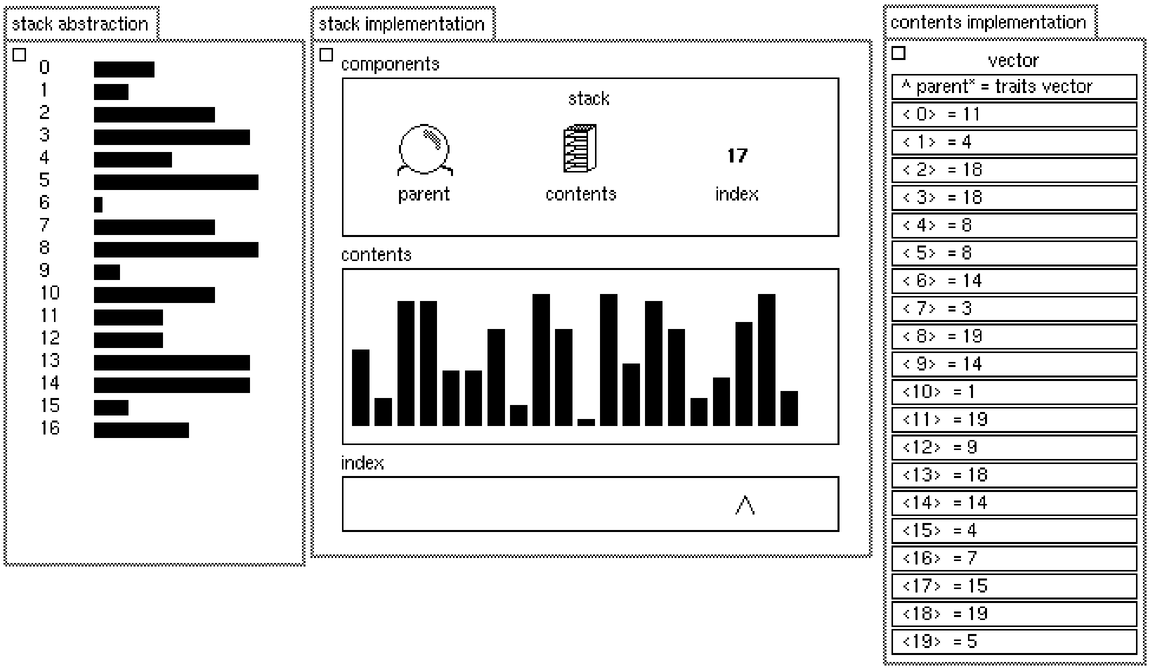

Figure 1: A Stack Abstraction and Implementation

a data structure constructed of records and pointers using a "box-and-arrows" diagram, and a programming environment could display the intermodule dependencies within a program as a graph. These views are typically updated when requested by the user, rather than continuously.

The difference between these categories is one of the level of *abstraction*. Graphical debugging works at the language level, while algorithm animation displays high level features of the program. Note that abstraction in this sense does not concern simply the amount of detail a visualisation contains. Rather, we consider the abstractions within a program to be the important ideas in the design of that program.

Figure 1 shows how different views, at multiple levels of abstractions, can be drawn for a simple example program (the views in this figure were created by our Tarraingím system). The stack abstraction view shows a push down stack of integers. Each stack element is displayed by the length of a horizontal bar. The stack currently contains seventeen elements, with the top element (numbered 0) drawn at the top of the view.

The stack implementation view shows how the stack is implemented — that is, it displays the stack at a lower level of abstraction than the stack abstraction view. This view has three subordinate views. The first of these, labelled components, contains icons for the main components of the implementation: a parent which supplies the stack's operations, an array holding the stack's contents, and an integer index. The other two views display these components. The contents view displays the contents array as another bar graph, although in this view the height of the bar gives the value of the array element, and elements are drawn from left to right. The index view illustrates the value of the index component as an index into the contents array.

The contents implementation view shows the underlying vector that implements the stack's contents array. This vector also contains a parent component, and components for each array element.

Several interesting features of the stack implementation can be gleaned by comparing these views. For example, the value of the index variable is the size of the stack. The contents array contains twenty elements (the stack only seventeen), so presumably when an element is removed from the stack it is not removed from the array, but the index is adjusted so that it is inaccessible to the stack operations. This can be seen quite clearly by watching the evolution of the display as elements are added to and removed from the stack. Finally, the order of items in the stack is the reverse of that in the array. The stack is "pushed down" the screen, so that the element most recently added is on the top of the stack. Elements are stored in the array in its natural order (drawn left to right), so that the top stack element can be anywhere in the array (actually one element to the left of the index), and the first (leftmost) array position holds the bottom stack element. To traverse the stack in its natural order, the portion of the array containing valid stack elements must be traversed in reverse order.

The views in Figure 1 also illustrate another important point about abstractions in programs. Any abstraction in a program can be seen from two different perspectives: its *interface* and its *implementation*. Seen from the perspective of its interface, an abstraction appears as a single unit which can be understood in isolation. An abstraction's implementation describes how the abstraction's interface is realised.

In Figure 1, the stack abstraction and stack implementation views present the interface and implementation perspectives of the stack abstraction respectively, and the stack's contents component view and the contents implementation view present the corresponding perspectives of the stack's component array.

To understand a program, a programmer must understand the hierarchy of abstractions and implementations from which it is constructed. A program visualisation system should therefore be able to display both perspectives of abstractions at any level in the program's design. We call a program visualisation system designed to highlight abstractions an *abstract* program visualisation system, and the Tarraingím system is our research prototype of such a system.

3 Explicit Abstractions

A program's abstractions must be embodied somewhere within the program. If a program uses a stack, it must contain an implementation of the stack, that is, data structures to contain the stack's elements, and statements to implement operations to push and pop data, determine if the stack is full, and so on. Unfortunately, the relationship between an abstraction and its implementation is not necessarily obvious. For example, a stack may be implemented by several blocks of memory and a pointer variable manipulated by several widely separated parts of the target program.

For this reason, program visualisation systems which display abstractions (typically algorithm animation systems) need information about the design of the target program to identify the important abstractions in the program's design, and to recover information from the implementations of those abstractions in the target program. This information is not necessarily explicit in a program's text, or in a record of a program's execution. In practice, abstractions in the target program can be identified using several techniques: the target program can be annotated to identify its abstractions and the interesting events in its execution (Brown, 1988a; Bentley and Kernighan, 1991), or procedures (Myers, 1983; Laffra and Malhotra, 1994) or mapping rules (Matsuoka, Takahashi, Kamada and Yonezawa, 1992; Cox and Roman, 1994) can be written to extract this information from the program. Although they differ in detail, the process to construct an abstract visualisation is roughly the same for each technique. A programmer must inspect the target program and understand how the abstractions to be visualised are implemented. The visualisation then is defined in terms of this implementation. We call this a *bottom up* visualisation.

Our approach is to base visualisations on abstractions which are *explicit* in the target program, and then to construct displays *top down* — working from the definitions of abstractions to their implementations. In this way, a program visualisation system can simply inspect the program to determine the abstractions it contains. The information required by views of these abstractions can be acquired by relying upon the explicit definition of the abstractions, rather than by reverse engineering the abstractions' implementations.

Consider the stack described above. If abstractions are represented explicitly in a program's structure, the stack can be detected simply by inspecting the abstractions the program contains. When a view needs information about the stack, the stack's definition can be consulted to determine how that information can be retrieved.

Objects Represent Design Abstractions

A top down approach will work provided that the abstract ideas on which a program is based are represented in the program in an easily identifiable way. This, of course, is precisely the basis of object oriented programming: an object oriented program is made up of a number of objects, each of which corresponds to an important concept in the program's problem domain (Meyer, 1988).

An object has an *interface* which is a list of operations which can be performed upon it. To be useful, an object must also have an internal description of how it is implemented in terms of lower level concepts. An object's implementation consists of a *representation* and a set of procedures. An object's representation contains the information required by each individual object, and the procedures perform the operations defined in the object's interface upon the representation.

The most important feature of object orientation is the separation of an object's interface and implementation. This is known as *information hiding* (also *privacy*, or *encapsulation*). The implementation of an object should only be available within its definition: in the rest of the program, the object should only be manipulable via operations in its interface.

An informal specification of the interface of a stack object is shown in Figure 2. Each operation supported by the object is listed, and the operation's behaviour described.

Stack Interface	
constructors	
new_stack	Create a new stack
mutators	
push(stack, e)	Push element e onto the stack.
pop(stack)	Remove the top element from the stack.
accessors	
top(stack)	Return the top element on the stack.
size(stack)	Return the number of elements in the stack.
isEmpty(stack)	Return true if the stack is empty.

Figure 2: Stack Interface

An object's operations can be grouped into several categories (Liskov and Guttag, 1986; Meyer, 1988). Operations which create a new object (new in the example) are known as *constructors*; operations which change the value contained within an object (push and pop) are known as *mutators*; and operations which retrieve information from an object but do not change its value (top, size, and isEmpty) are known as *accessors*.

Visualising Objects as Abstractions

Debuggers (whether textual or graphical) retrieve low level information directly from the target program's memory space. By analogy, abstract data can be retrieved from objects by calling accessor operations which return that information. Because the data returned is abstract and independent of a particular implementation, the program visualisation system does not have to read the target program's memory directly, and in particular, it does not have to reinterpret any implementation data structures. For this approach to work, the program visualisation system must be able to call procedures defined in the target program when information is required, rather than simply inspect its memory.

Debuggers monitor dynamic behaviour either by single stepping the target program, or by tracing calls to procedures defined in the program. The behaviour of an object can be monitored by a similar analogy; rather than indiscriminately monitoring procedures, we monitor those procedures implementing that object's operations. Because object's implementations are hidden behind their interfaces, all operations upon abstractions must be performed through their interfaces. By monitoring the object's public operations, we are assured of monitoring all the computation performed in the context of the design abstraction that object represents.

Since an object's implementation is a collection of data and associated procedures, implementation views of objects can be produced by inspecting the state of the representation data and monitoring the procedures.

The encapsulation promoted by object orientation, especially when supported by an object oriented programming language, ensures that objects can be considered in isolation. In particular, all operations upon an abstraction can be detected by monitoring the interface of the object representing that abstraction. For this reason, we based the design of Tarraingím on the object oriented programming paradigm, and in particular the object oriented programming language Self. Tarraingím's model of program animation is based on the following principle:

The *pictures* we need to draw correspond to the *abstractions* in the *design* which are the *objects* in the *program*.

4 An Object Oriented Framework

As well as visualising programs written in an object oriented manner, Tarraingím's design is also based on objects. The essence of the Tarraingím system is an *object oriented framework* (Johnson and Foote, 1988; Johnson, 1992) consisting of three *subsystems*: Monitoring, Strategy, and Display (see Figure 3). The Tarraingím system itself consists of libraries of objects which are used within the framework, providing both a program visualisation environment and an extensible program visualisation kit.

	Tarraingím		
Framework (subsystems)	Monitoring	Strategy	Display
Objects	controller encapsulator	watchers	views
	events		

Figure 3: Subsystems

Most of the objects in Tarraingím's framework belong to one of the three subsystems.

Monitoring

The monitoring subsystem monitors the target program to gather the information about the program's *actions* (message receipts and variable accesses and assignments) required by the rest of the visualisation system.

Two main kinds of objects are used in this subsystem. The target program is actually monitored by encapsulator objects (Pascoe, 1986; Foote and Johnston, 1989). In an earlier paper (Noble and Groves, 1992), we discuss the implementation of encapsulators used in Tarraingím. The interface to encapsulators is provided by controller objects, which also route the information encapsulators gather to the rest of the system. The rest of Tarraingím depends upon controllers rather than encapsulators. In this way, Tarraingím is independent of the particular technique used to monitor the target program.

Strategy

The strategy subsystem is responsible for linking the monitoring subsystem to the display subsystem. This is primarily carried out by watcher objects. Initially, this involves requesting that the monitoring system monitor the actions of objects which are to be displayed in the watcher's associated view. Once the target object's actions are being monitored, watchers translate these actions into *changes* to be applied to the view. Watchers also handle *callbacks* sent from the display subsystem, and translate them into message sends to objects in the target program.

Different kinds of watchers embody different strategies for monitoring objects and performing the mapping. For example, **interfaceWatchers** notify views about messages sent to an object from other objects in the program, **localWatchers** notify views about changes in an object's internal representation, and **traceWatchers** notify views about all the messages handled by an object's implementation.

Display

The display subsystem has three tasks, of which the most important is to draw the graphical images making up the visualisations. It also provides the user interface for the rest of Tarraingím, and handles any user input to the visualisations. The main object in this subsystem is the view. A view implements a display of a particular object in the target program. Different kinds of views provide different displays – for example, a treeView displays a tree, while a barGraph view displays a graph.

Views receive information about the target program in two ways. First, they receive changes from the monitoring subsystem, which describe the progress of the target program's execution. For example, when an object receives a message, a watcher will relay a change describing that event to its associated view. Second, they may send callbacks to objects in the program, which may be either queries to determine the state of objects in the program, or user commands to alter the objects. For example, when a view needs information about an object, it sends a callback to its associated watcher which in turn sends a message to the object to retrieve the required information.

Events

Event objects are essentially typed data packets. They are used to carry information around the Tarraingím framework, and do not belong to any particular subsystem.

Most events are *up events*, which are sent from controllers to watchers to describe the actions of the target program. Watchers forward events to views to describe the changes required in the view. Views send *down events* which are eventually routed via watchers to the objects in the program. Down events are used to implement callbacks.

Framework Arrangement

The dynamic arrangement of a simplified version of Tarraingím's framework is illustrated in Figure 4. Objects are arranged in a pipeline. An encapsulator monitors an object in the program. A controller packages the information gathered by encapsulators into event objects, and sends them to a watcher. The watcher processes the events before finally sending them to the view, which produces the graphical output. The view may send requests or commands to its target object. These requests are packed as callback events, and are passed in the opposite direction along the pipeline. Some views are able to accept user input, and this is also handled by views sending callback events to watchers.

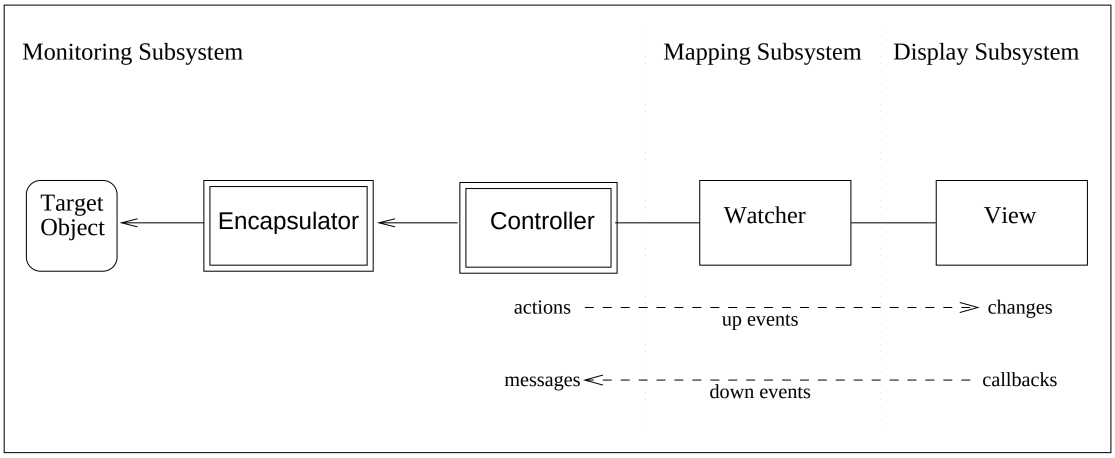

Figure 4: Tarraingím's Framework

The watcher, view and event objects are abstract. Tarraingím's library includes many different concrete implementations, and a programmer can write new versions of these objects to extend the system. The controller and encapsulator objects are concrete, and suitable for all types of visualisation. They are not normally replaced.

Multiple objects can be displayed using several parallel pipelines. Each visualised object will have its own encapsulator, controller, watcher, and view. The pipeline can be generalised into a tree, allowing multiple views to display one object. Each view has its own watcher, while all views and watchers monitoring a single object share a single controller-encapsulator pair.

A view can contain one or more independent *subviews*. These may display different representations of the same object, or information about several related objects. Similarly, watcher objects can rely on *subwatchers* when implementing complex strategies. Subwatchers can also be used to allow a single view to monitor multiple objects.

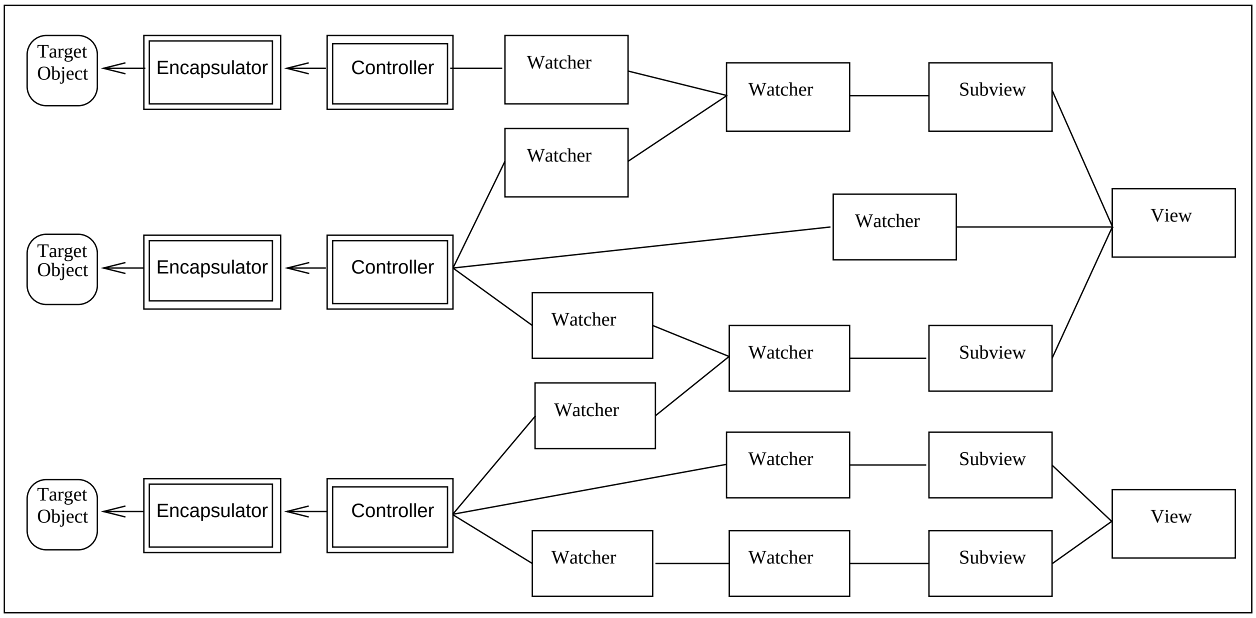

Figure 5: Multiple Views and Multiple Watchers

Figure 5 illustrates the overall scheme. The displays are produced by a forest of views, each of which can use a tree of watchers. The leaves of the watcher trees are attached to controllers and encapsulators associated with the objects they are visualising.

5 Examples

Using Tarraingím's Framework

Simple displays can be implemented with only one view and one watcher. For example, Figure 6 shows a Tarraingím view which displays the structure of the stack abstraction view from Figure 1. The view in Figure 6 is a *reflexive* view, since it illustrates Tarraingím itself, rather than the target program. Icons representing the stack and the stack's controller appear to the left of the reflexive view, the watcher is drawn in the middle of the reflexive view, and the actual view object which produces the view shown in Figure 1 is to the right. From the view structure view, we can see that the watcher is actually an interfaceWatcher, which monitors an object's interface, and that the view is a barGraph, which draws a vertical bar graph. Note that reflexive views do not display encapsulators, since encapsulators are hidden within the monitoring subsystem and are always accessed via controllers.

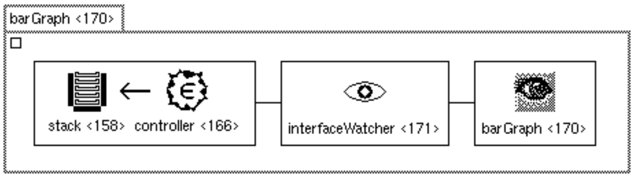

Figure 6: Structure of stack abstraction view

Figure 7 shows another view of the stack abstraction, and another reflexive view displaying the structure of that view. In this Figure, the stack's contents are displayed as a horizontal list, rather than as a bar graph. From the reflexive view in Figure 7, we can see that this display is produced by an hCollView, and that the hCollView is connected to the stack object by another interfaceWatcher. This illustrates the advantage of separating the responsibility for processing events from the target object (handled by the watcher) from the responsibility for producing graphical output (handled by the view). In spite of their different appearances, both barGraphs and hCollViews require the same information about the objects they are displaying, and this information can be provided by the same type of watcher.

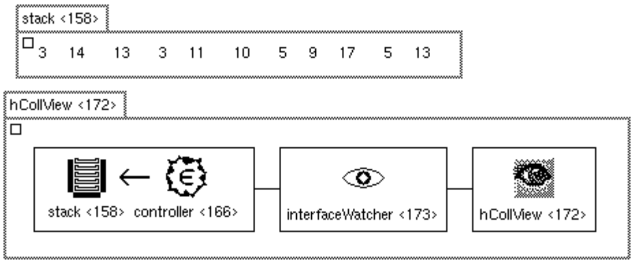

Figure 7: Alternative view of stack object

Figure 8 shows a trace of recent operations performed by the stack, that is, of messages received by the stack object, and a reflexive view display showing the trace view's structure. The trace is displayed by a tracer view object, which simply lists the events it receives from its watcher. Again, an interfaceWatcher is used so that the trace view displays messages sent to the stack object's interface.

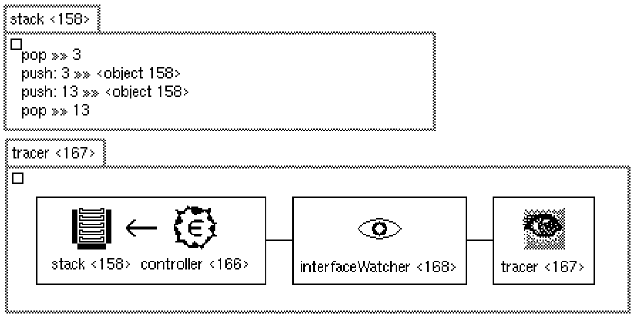

Figure 8: Interface trace of stack object

Figure 9 shows another trace view (like that in Figure 8), but this view displays operations internal to the stack, as well as operations at the stack's interface. Figure 9 illustrates a trace of the implementation of a pop operation: the stack is checked to ensure that it is not empty, the stack's index is decremented, and then an element is returned from the stack's contents array. The reflexive view shows that this implementation trace is implemented by the same kind of tracer view as the abstract trace from Figure 8, however, it is connected to the target object by a traceWatcher, rather than an interfaceWatcher. Since the watcher determines the events a view receives, by changing watchers we can reuse view objects to produce different visualisations.

More complex displays often require multiple views. Consider the reflexive views from the examples above. Each individual icon is implemented by an independent view and watcher pair. Similarly, each array element in the contents implementation view in Figure 1 is displayed by an independent view.

Visualising a Parser

The Illustrated Compiler (Andrews, Henry and Yamamoto, 1988) visualised a PL/0 compiler (Wirth, 1976). Both the compiler and visual displays were implemented specially in Lisp, and the compiler was modified to generate information to keep the displays correct. We have implemented a lexical analyser and top down parser for a PL/0-like language in SELF (the Illustrated Compiler also contained a pseudocode generator, and an interpreter), and have used views from Tarraingím's library along with a few custom views to visualise this parser.

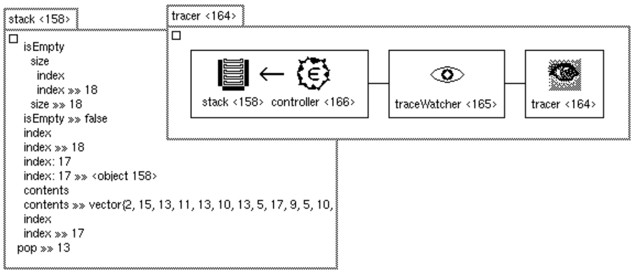

Figure 9: Internal trace of stack object

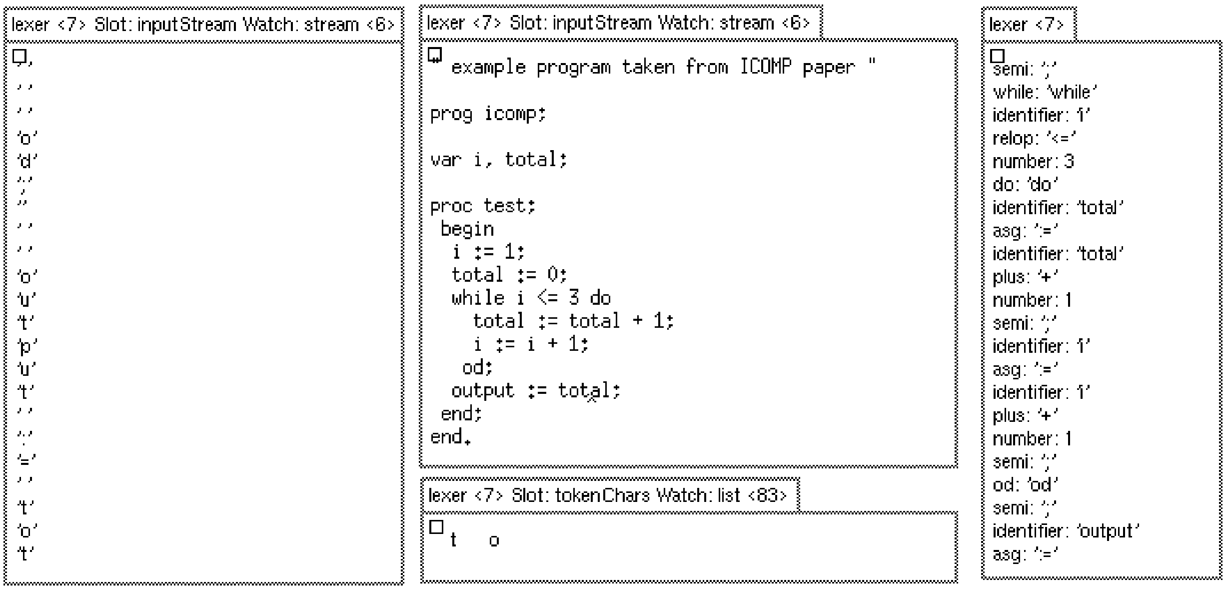

Figure 10: The inputStream

Our parser's lexical analyser (lexer) is displayed in Figure 10. The left and right hand views in the Figure are Tarraingím's standard trace views. In the figure, these views are not displaying message traces. Instead, they provide traces of the inputStream's and lexer's output. These views use a custom nextWatcher, which sends its associated view when the target object finishes executing the **next** message. SELF streams respond to the **next** message by returning their next element, so that these views provide a trace of the values returned by successive calls to **next**. Figure 10 thus displays the parser's lexical analysis as an abstract stream transformation: the input to the lexer (in the left hand view) is a stream of characters, and the output (in the right hand view) is a stream of tokens.

The two centre views in Figure 10 display further information about the lexer's input processing. The centre lexer inputStream view displays the text of the program being parsed. The lexer tokenChars view displays a buffer in the lexer accumulating characters from the input stream into the next token to be returned by the lexer. The tokenChars view is implemented by the hCollView illustrated in Figure 7.

Finite State Machine

The main view we built specially for the parser visualisation is illustrated in Figure 11. This view displays the finite state machine used to direct the lexer.

State machine states are drawn as named rectangles; the start state (named start) is drawn with a triangle

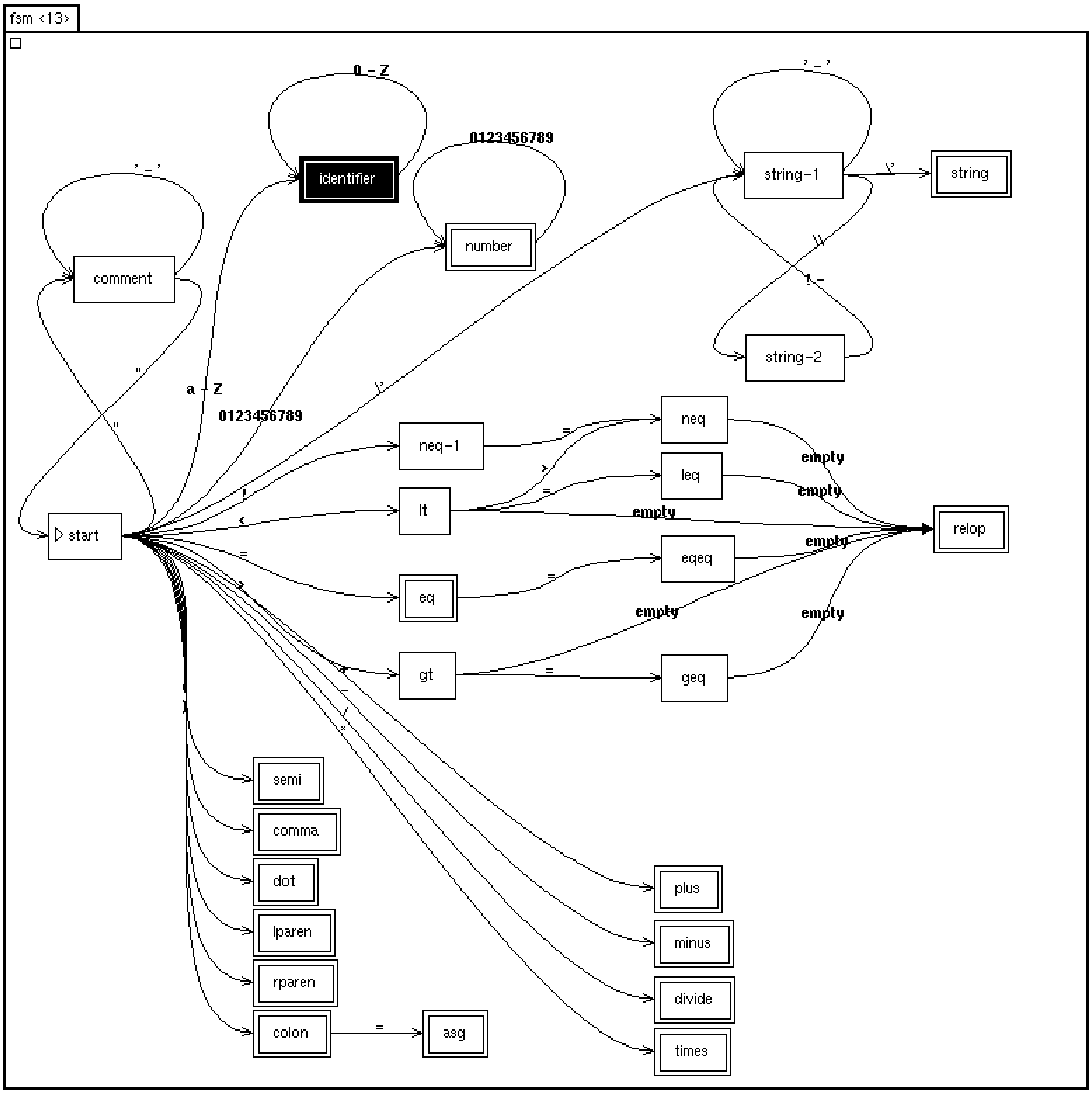

Figure 11: Finite State Machine

before its name, and terminal states are drawn with a double border. Transitions are represented by arrows linking states and are labelled with the input characters or character ranges which cause the transitions to be taken. The current state of the FSM is shown by a state drawn in reverse video. The FSM is shown in the state identifier, which is a final state.

In this view, each FSM state is displayed by a fsmStateView, which uses a localWatcher to monitor the corresponding FSM state, and sends callbacks to determine whether it is a special (start or terminal) state. The main FSM view draws the transition lines linking states, using auxiliary objects to position the splines and labels. State views are laid out manually by the user, although any particular layout can be stored and recalled by the FSM view.

Grammar Rules

The parser's grammar rules are represented as trees of grammar nodes. Figure 12 shows the inheritance hierarchy of grammar node objects — the basic nonNode and termNode representing terminal and nonterminal symbols, and the seqNode, selNode, it0Node, and it1Node representing sequence, selection, and iteration (zero or more times, and one or more times, respectively). This view is drawn by a treeView from Tarraingím's library: each individual node is an independent view.

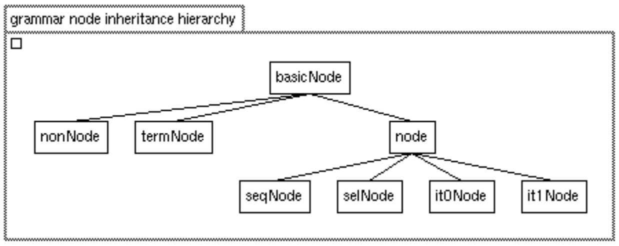

Figure 12: Grammar Node Inheritance Hierarchy

Figure 13 illustrates the actual grammar rule for while statements. In Figure 13, nodes labelled "?" are selection nodes, and nodes labelled "*" are it0Nodes. The parser recursively traverses these grammar nodes, building an abstract syntax tree. The view in Figure 13 is also constructed using Tarraingím's tree view.

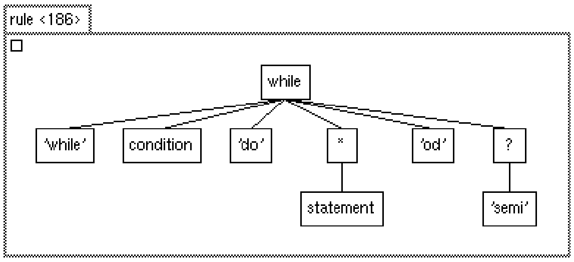

Figure 13: Rule for While statement

Parsing

The parseTree views in Figure 14 display the abstract syntax tree being constructed by the parser, at two different magnifications. The large displays are created by the generic tree view mentioned above. The smaller display is essentially the same view, but configured differently: in particular, boxViews, which appear as small rectangles, are used to display the nodes in place of treeNodeViews used in the larger view.

Summary

While our example parser and its visualisation here are less complete as those of Andrews et al. (1988), we consider that this is a matter of the time and effort expended, rather than a shortcoming of the visualisation approach itself. Our display of the lexer and parser shows that our abstraction-based approach to visualisation can produce displays of similar quality to hand-crafted techniques such as those used in the Illustrated Compiler. More importantly, only a few of the views presented in this section were written specially for this example: the majority were taken from Tarraingím's view library.

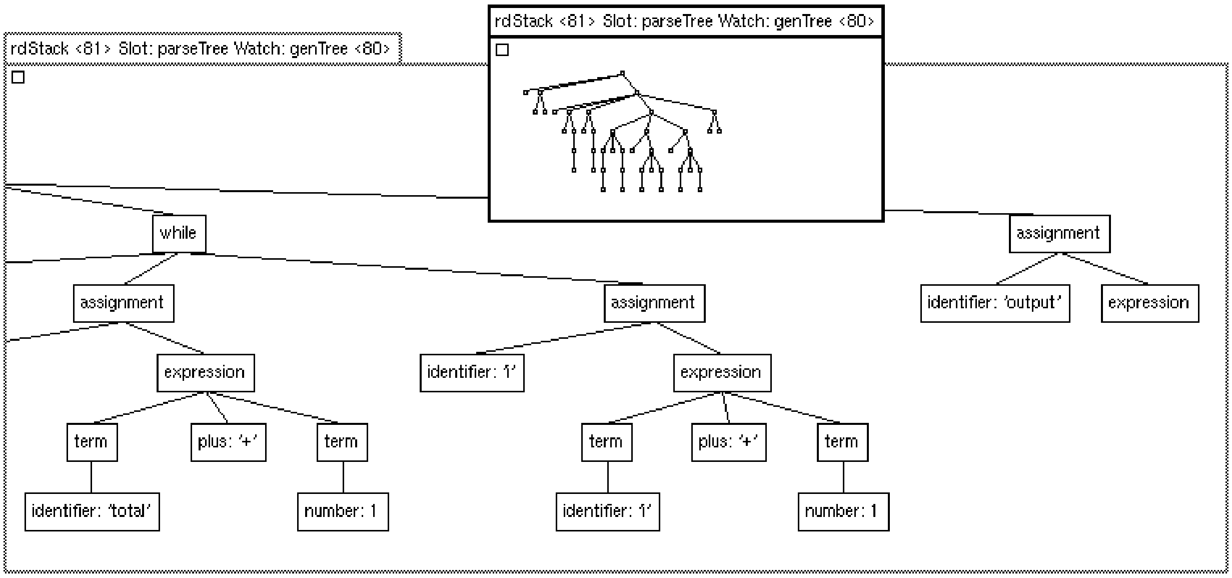

Figure 14: Parse Tree

6 Related Work

The first attempts at program visualisation were computer generated movies which illustrated the behaviour of a fundamental algorithms, for example Baecker and Sherman (1981). As we mentioned in Section 2, more recent work has been based mostly upon traditional languages, and is directed towards either low level debugging or high level algorithm animation. Myers (1990) and Price, Baecker and Small (1993) contain surveys and taxonomies covering both graphical debuggers and algorithm animation systems. In this section we focus upon the use of object orientation in program visualisation.

Graphical Debuggers Graphical debuggers use graphical user interface (GUI) techniques to illustrate the target program's data structure and control flow, in terms of the basic objects and operations provided by the target language. For example Amethyst (Myers, Chandhok and Sareen, 1988) can display all Pascal's data types including records, arrays, and pointer structures. The Object Visualizer (Pauw, Helm, Kimelman and Vlissides, 1993) introduces several interesting displays which present information about control flow, data structure, and program performance, but again these displays illustrate the program at the level of the programming language. Virtual Images (Vion-Dury and Santana, 1994) provides animated 3D displays of distributed object oriented programs. Graphical debuggers are now available commercially, for example, the Object Explorer (Beck, 1994) for Smalltalk, or Look! (West, 1994) for C++.

Some graphical debuggers are extensible, that is, they allow the user to define custom graphical displays. For example, Vick, a visualisation construction kit for Smalltalk (Böcker and Pawlitschek, 1992), includes a graphical editor which can be used to build a composite display by combining smaller display components. Similarly Cerno-II (Fenwick, Hosking and Mugridge, 1994), a graphical debugger for the object oriented Prolog dialect Snart, uses a declarative language to specify the displays to draw.

However the displays and abstractions are defined, the emphasis of all of these systems is still bottom up, as they derive the abstractions to visualise from the concrete implementation of objects, rather than the objects' abstract definitions.

Algorithm Animation Algorithm animation systems are the other end of the spectrum. They produce user defined views of programs at high levels of abstraction, but lack the low level display and control capabilities of graphical debuggers.

Annotation is the most common algorithm animation technique (Brown, 1988b; Bentley and Kernighan, 1991). The target program is annotated or instrumented with event markers, which are essentially calls to the program visualisation system. When the target program runs, the event markers notify the visualisation system about the target program's actions. Annotation is therefore implemented by editing the program's code, and this is its greatest disadvantage: to annotate a program one must make semi-permanent changes to it. Event markers are very flexible monitoring tools because they can be positioned precisely to capture important parts of the program's execution, but this incurs the complementary disadvantage because annotating a program requires an intimate knowledge of its structure and implementation.

Mapping based algorithm animation systems (Roman, Cox, Wilcox and Plun, 1992; Matsuoka et al., 1992; Henry, Whaley and Forstall, 1990) are the main alternative to annotation based systems. Like annotation systems, mapping systems also work bottom up — they monitor the low level actions of the target program and then use inference rules to determine the higher level changes from the program's actions. Mapping can in principle avoid many of the problems of annotation — in particular, the target program is not annotated or modified. The program can be changed without affecting the visualisation, as there are no annotations which must be preserved.

Several algorithm animation systems have been built using object orientation. For example, the Animation Kit (London and Duisberg, 1985) is an annotation based system built on top of Smalltalk's MVC interface framework and dependency mechanism (Goldberg and Robson, 1983). Duisberg (1987) describes another annotation based system built in Smalltalk, which allowed simple visualisations to be built by direct manipulation, without any textual programming. Haarslev and Möller (1992) describes a visualisation system which uses the reflexive facilities of Clos to link programs to views.

Programming Environments It is also possible to consider as related the programming environments of languages such as Smalltalk (Goldberg, 1983) or SELF's own user interface (Chang and Ungar, 1990). Although these interfaces are undoubtedly more polished than the current Tarraingím system, they provide only low level views of the objects comprising the program.

Probably the most advanced visual programming environments are those for visual programming languages (VPLs) such as Prograph (Cox, Giles and Pietrzykowski, 1989), or Serius (Landay, 1991). These languages have a visual syntax, rather than a traditional textual syntax: a VPL program is an arrangement of figures on a page, rather than a stream of text. However, displaying the program's visual syntax is generally the limit of the visualisation facilities provided by VPLs, and these views still display only the syntax of the program.

User Interfaces It is interesting to compare Tarraingím with user interface systems such as Smalltalk's MVC (Goldberg and Robson, 1983; Lalonde and Pugh, 1991). Although not designed as a user interface framework, we have experimented with using Tarraingím to construct user interfaces — in particular, Tarraingím has been used to build its own interface.

With regards to MVC, the views correspond to the MVC views encapsulated objects to MVC's model, and Tarraingím's controller to MVC's implicit dependence relationship. Tarraingím's watchers can act as the pluggable adaptors used in more recent versions of MVC (Lewis, 1995), although Tarraingím's watchers also have other responsibilities, such as planning the way an object is to be monitored. Currently there is no component corresponding to MVC's controller: MVC's controller's input handling functions are carried out by Tarraingím's views. Tarraingím's event objects have no explicit parallel in MVC.

Tarraingím can also be usefully compared to constraint based approaches such as those used in Animus (Borning and Duisberg, 1986) or Garnet (Myers, Guise, Dannenberg, Vander Zanden, Kosbie, Pervin, Mickish and Marchal, 1990). Generally such systems treat objects as collections of slots with values (Myers, Giuse and Zanden, 1992) rather than abstract encapsulations of state and behaviour. Constraint based systems update displays of objects by propagating changes in slot values, while Tarraingím monitors objects' interfaces and sends events to indicate changes.

7 Discussion

Objects and Abstractions

The idea of using objects to represent a program's abstractions for visualisation is not new. Section 6 referred to several visualisation systems which used objects, but these produced views bottom up, working from object's implementations rather than their interfaces.

Of course, for Tarraingím's approach to work, the objects represented by the target program's structure must represent the abstractions the programmer considered important. However, these abstractions are not necessarily those the user wishes to see. For example, the user may wish to visualise the lexical analysis performed by a compiler. The design of the target compiler may distribute the lexical analysis throughout various parts of the compiler, and may not incorporate a separate lexer to visualise. In general, a program can be designed in more than one way. The programmer may have chosen one design (e.g., a parser without a separate lexer), while the user of the animation system may assume another.

The advantage of using the target program's object oriented structure to capture program abstractions is that information about these abstractions can be obtained easily. This advantage applies only to those abstractions explicit in the program's design. Program visualisation implies a relationship between the target program and the resulting illustrations. The user of a program visualisation system is ultimately investigating a real, concrete, implemented program.

In developing the BALSA algorithm animation system, Brown (1988b) also considers using objects to represent abstractions, and monitoring objects to detect changes, but dismisses this option in favour of annotating the target program. Brown rejects using objects for two main reasons. First, the target programs in which he is interested, examples from standard textbooks, are not written in that style. Second, inserting annotations is less effort than restructuring the target program. This rejection is a design trade-off related to the different application domains of Brown's algorithm animation and our program visualisation. Algorithm animation addresses small programs which contain great procedural complexity: the detailed operations occurring in these programs are very important in their visualisation. With abstract program visualisation systems, we are attempting to address moderately sized programs made up of many abstractions, where each abstraction, considered individually, is not particularly complex. Presenting serviceable views of all the abstractions in the program is more useful to the practicing programmer than presenting very detailed views of any particular abstraction.

Reuse

As illustrated by the examples in Section 5, Tarraingím's object oriented framework, and the use of objects to capture the abstractions in the target program, supports the reuse of visualisations (i.e. views and watchers) and allows new views to be built by composition. We have found that this reuse is quite possible within a single program or a single library. For example, the reuse of collection views to display the stack (Figure 6 and Figure 7) relies upon the stack conforming to the standard interface of collection objects in the SELF library. Similarly the use of the stream view to display the parser's input stream and token stream in Figure 10 depends upon the common stream interface used within the design of the parser example program.

It would be interesting to investigate the possibilities for reuse across different programs within the same domain — that is, across different programs which perform related tasks and can be expected to share some abstractions. For example, (Agesen, 1994) describes Mango, an alternative parser generator also written in Self. Ideally, the views we constructed to display our example parser would be able to be reused to visualise a Mango parser, as both programs operate in the same domain.

Conclusion

We have described the design of the Tarraingím program visualisation system. Tarraingím benefits from object orientation in two ways. First, objects are used to organise the target program, so that the target program's abstractions are made explicit in its structure. Tarraingím has successfully visualised small programs at multiple levels of abstraction, without the implementation of those abstractions being described separately to the visualisation system. Second, Tarraingím's architecture is based upon an object oriented framework. We have found this provides a flexible architecture for producing multiple views, and have been able to create new visualisations by reusing existing components. We are continuing to experiment with Tarraingím, and program visualisation based upon objects, to support the visualisation of large and complex programs.

Acknowledgements

This work is supported by the Victoria University of Wellington Internal Grants Committee, the University Grants Committee, IBM New Zealand, and the Centre for Object Technology, Applications and Research at the University of Technology, Sydney. We would like to thank the SELF group at Stanford University and Sun Microsystems Laboratories for their assistance.

References

AGESEN, O. (1994): Mango - A parser generator for Self, Technical Report TR-94-27, Sun Microsystems Laboratories.

ANDREWS, K., HENRY, R. and YAMAMOTO, W. (1988): The design of the UW illustrated compiler, in *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implmentation.*

BAECKER, R. M. and SHERMAN, D. (1981): *Sorting Out Sorting*, 16 mm colour sound film. Shown at ACM SIGGRAPH Conference.

BECK, K. (1994): Pictures from the object explorer, Technical report, First Class Software, Inc.

BENTLEY, J. L. and KERNIGHAN, B. W. (1991): A system for algorithm animation, *USENIX Computing Systems* **4**(1).

BÖCKER, H.-D. and PAWLITSCHEK, M. (1992): VICK: a visualization construction kit, *Journal of Object-Oriented Programming* .

BORNING, A. and DUISBERG, R. (1986): Constraint based tools for building user interfaces, *ACM Transactions on Graphics* **5**(4).

BROWN, M. H. (1988a): *Algorithm Animation*, ACM Distinguished Dissertation, MIT Press.

BROWN, M. H. (1988b): Perspectives on algorithm animation, in *Human Factors in Computing Systems (ACM CHI Conference Proceedings).*

CHANG, B.-W. and UNGAR, D. (1990): Experiencing SELF objects: An object-based artificial reality, in U. HÖLZLE, ed., *The Self Papers*, The Self Group.

COX, K. C. and ROMAN, G.-C. (1994): A characterization of the computational power of rule-based visualization, *Journal of Visual Languages and Computing* **5**(1).

COX, P. T., GILES, F. R. and PIETRZYKOWSKI, T. (1989): Prograph: a step towards liberating programming from textual conditioning, in *IEEE Workshop on Visual Languages.*

DUISBERG, R. (1987): Visual programming of program visualizations, in *Proc. IEEE Visual Languages Workshop.*

FENWICK, S., HOSKING, J. G. and MUGRIDGE, W. B. (1994): Cerno-II: A program visualization system, Technical Report 87, Auckland University.

FOOTE, B. and JOHNSTON, R. E. (1989): Reflective facilities in Smalltalk-80, in *OOPSLA Proceedings.*

GOLDBERG, A. (1983): *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley.

GOLDBERG, A. and ROBSON, D. (1983): *Smalltalk-80: The Language and its Implementation*, Addison-Wesley.

HAARSLEV, V. and MÖLLER, R. (1992): Visualisation and graphical layout in object-oriented systems, *Journal of Visual Languages and Computing* **3**(1).

HENRY, R. R., WHALEY, K. M. and FORSTALL, B. (1990): The University of Washington Illustrating Compiler, *ACM SIGPLAN Notices* **25**(6).

JOHNSON, R. E. (1992): Documenting frameworks using patterns, in *OOPSLA Proceedings*.

JOHNSON, R. E. and FOOTE, B. (1988): Designing reusable classes, *Journal of Object-Oriented Programming* **1**(2).

LAFFRA, C. and MALHOTRA, A. (1994): Hotwire — a visual debugger for C++, in *USENIX C++ Conference*.

LALONDE, W. and PUGH, J. (1991): *Inside Smalltalk*, Vol. 2, Prentice-Hall.

LANDAY, J. A. (1991): Tools review: Serius—a visual programming environment, *Journal of Visual Languages and Computing* **2**(3).

LEWIS, S. (1995): *The Art and Science of Smalltalk*, Prentice-Hall.

LISKOV, B. and GUTTAG, J. (1986): *Abstraction and Specification in Program Development*, MIT Press/McGraw-Hill.

LONDON, R. and DUISBERG, R. (1985): Animating programs using Smalltalk, *IEEE Computer* **18**(8).

MATSUOKA, S., TAKAHASHI, S., KAMADA, T. and YONEZAWA, A. (1992): A general framework for bi-directional translation between abstract and pictorial data, *ACM Transactions on Information Systems* **10**(4).

MEYER, B. (1988): *Object-oriented Software Construction*, Prentice Hall.

MYERS, B. A. (1983): Incense: A system for displaying data structures, in *SIGGRAPH Proceedings*.

MYERS, B. A. (1990): Taxonomies of visual programming and program visualization, *Journal of Visual Languages and Computing* **1**(1).

MYERS, B. A., CHANDHOK, R. and SAREEN, A. (1988): Automatic data visualization for novice Pascal programmers, in *Proc. IEEE Visual Languages Workshop*.

MYERS, B. A., GIUSE, D. A. and ZANDEN, B. V. (1992): Declarative programming in a prototype-instance system: Object-oriented programming without writing methods, in *OOPSLA Proceedings*.

MYERS, B. A., GUISE, D. A., DANNENBERG, R. B., VANDER ZANDEN, B., KOSBIE, D. S., PERVIN, E., MICKISH, A. and MARCHAL, P. (1990): Garnet: Comprehensive support for graphical, highly interactive user interfaces, *IEEE Computer* **23**(11).

NOBLE, R. J. and GROVES, L. J. (1992): Tarraingím — A Program Animation Environment, *New Zealand Journal of Computing* **4**(1).

PASCOE, G. A. (1986): Encapsulators: A new software paradigm in Smalltalk-80, in *OOPSLA Proceedings*.

PAUW, W. D., HELM, R., KIMELMAN, D. and VLISSIDES, J. (1993): Visualizing the behavior of object-oriented systems, in *OOPSLA Proceedings*.

PRICE, B. A., BAECKER, R. M. and SMALL, I. S. (1993): A principled taxonomy of software visualization, *Journal of Visual Languages and Computing* **4**(3).

ROMAN, G.-C., COX, K. C., WILCOX, D. and PLUN, J. Y. (1992): Pavane: a system for declarative visualization of concurrent computation, *Journal of Visual Languages and Computing* **3**(2).

SHILLING, J. J. and STASKO, J. T. (1994): Using animation to design object-oriented systems, *Journal of Object Oriented Systems* **1**(1).

STASKO, J. T. (1990): Simplyfing algorithm animation with TANGO, in *Proc. IEEE Visual Languages Workshop*.

UNGAR, D. and SMITH, R. B. (1987): SELF: the Power of Simplicity, in *OOPSLA Proceedings*.

VION-DURY, J.-Y. and SANTANA, M. (1994): Virtual Images: Interactive visualization of distributed object-oriented systems, in *OOPSLA Proceedings*.

WEST, A. (1994): Making a case for animating C++ programs, *Dr. Dobb's Journal* .

WIRTH, N. (1976): *Algorithms + Data Structures = Programs*, Prentice-Hall.