VICTORIA UNIVERSITY OF WELLINGTON

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

Tarraingím – A Program Animation Environment

R. James Noble and Lindsay J. Groves

Technical Report CS-TR-91/2
October 1991

Abstract

Program animation is the use of computer graphics and animation techniques to visualise the behaviour of an executing program. This paper describes a program animation system being developed at Victoria University. This system (called Tarraingím from the gælic for *to draw*) is designed to include facilities of both graphical debuggers and algorithm animation systems; that is, Tarraingím will provide both low level views of program components and user-defined higher level views. As well as discussing the motivation and general architecture of Tarraingím, this paper discusses some of the technical problems arising in the implementation of such a system, and our solutions to them.

Publishing Information

This report appeared in the *Proceedings of the 12th New Zealand Computer Conference*, Dunedin. 14–16 August 1991.

Author Information

James Noble completed a B.Sc. (Honours) in computer science at Victoria University in 1988. He is currently investigating program animation systems using Tarraingím as a research model. He hopes to complete a Ph.D. thesis before the end of the millennium.

Lindsay Groves has a BSc in Mathematics from the University of Auckland and an MSc in Computer Science from Massey University. He worked for the Applied Mathematics Division, D.S.I.R., from 1974 to 1976. Since 1976 he has lectured in Computer Science, at Massey University, Waikato University, and since 1985 at Victoria University of Wellington. His main interests are in program correctness, automated reasoning, logic programming and software engineering.

1 Introduction

Program animation is the use of computer graphics and animation techniques to visualise the behaviour of an executing program. Such systems may be used by programmers in debugging and understanding programs and also in teaching programming.

The simplest program animation system is a graphical debugger, several of which are commercially available. These systems provide a sophisticated environment for controlling the execution of a program being debugged, and can also draw illustrations of the data structures used within the program. However, graphical debuggers have no knowledge of a program other than that explicitly contained within the program's source code. Thus, the only things that can be displayed are those primitive structures provided by the programming language, and they can only be drawn in ways designed into the debugger program.

More complicated algorithm visualisation tools also exist [Raeder, 1985, Brown, 1988, Stasko, 1989, Myers, 1990, Haarslev and Möller, 1990, Henry *et al.*, 1990]. These are designed to animate illustrations of algorithms one might find in programming text books. Generally, the animations can be tailored to the particular target algorithm. Rather than working at the level of the programming language, these systems are controlled by application-level clues either inserted manually into the program, or computed by an inferencer. This allows graphical displays at the application level, rather than in terms of the implementation language. However, these systems are oriented towards providing higher level views; they do not have the low-level display and control capabilities of the graphical debuggers.

Tarraingím is written in the object-oriented programming language, SELF, and animates programs written in SELF [Ungar and Smith, 1990, Hölzle *et al.*, 1991]. The Tarraingím system allows a programmer to observe the execution of their program. Because Tarraingím understands the basic structure of the language, it can display language-level views of objects. Tarraingím can monitor a program as it runs, and is able to keep displays updated, or produce execution traces and generate profiles of a program's execution. Finally, Tarraingím allows the programmer to define higher-level views of objects, and use the supplied monitoring facilities to keep these views correct as the program executes.

In the next two sections we introduce the SELF programming language, describe a small program in SELF and show how Tarraingím can be used to illustrate its execution. Next, we describe the some interesting points of the design and implementation of the system. Finally, we review our work and outline some possible directions for future work.

2 Programming in SELF

SELF is an object-oriented programming language developed by a group at Stanford lead by David Ungar. The syntax of SELF is based upon that of Smalltalk and, like Smalltalk, is strongly dynamically typed. Unlike Smalltalk, however, SELF is prototype-based, i.e. SELF does not use a class hierarchy in the definition of objects. Objects describe themselves, and may share state and behaviour with other objects through inheritance. The current implementation of SELF [Chambers *et al.*, 1990] uses a dynamic compiler and approaches half the speed of optimised C.

As with all object-oriented languages, a program in SELF consists of a set of objects and computations are invoked by sending messages to objects. Objects in SELF are collections of named slots, which can contain data objects or method objects (executable code). To allow objects to inherit state and behaviour, slots can also be designated as "parent" slots. Common state and behaviour can thus be factored out of most objects, into "traits" objects containing the shared information.

When a SELF object is sent a message, the object is searched for a slot which matches that message. If the slot contains a data object, the data object is returned; if the slot contains a method object the method is executed. If the search does not find a matching slot, then the object referred to by the parent slot is searched.

The example in Figure 1 shows an object object called "`prototypes trafficLight`" with two slots: one holding the colour (initialised to `'red'`) and one designating the object called "`traits trafficLight`" as its parent, from which it inherits four methods (`red`, `amber`, `green` and `cycle`).

```
traits trafficLight _Define:
       (|
       ''slots containing behaviour''
             red = (colour:  'red').
             amber = (colour:  'amber').
             green = (colour:  'green').
             cycle = (red.  amber.  green.  red).
       |)

prototypes trafficLight _Define:
       (|
       ''parent slot for inheritance''
             parent* = traits trafficLight.
       ''slot holding colour of the light''
             colour <- 'red'.
       |)
```

Figure 1: Example SELF program

3 Animating a SELF program with Tarraingím

Tarraingím can display a low level view of any SELF object. Figure 2 shows low-level views of several SELF objects, including traffic light prototype and traits objects similar to those shown in Figure 1. The displays show each slot in the object, and the values in these slots. This is equivalent to the display provided by the SELF user interface [Chang and Ungar, 1990]. However, the Tarraingím low-level views remain on screen as the program is executed, and are updated as the displayed objects change.

Tarraingím can display more low level information than just object slot values. Figure 3 shows the execution trace of messages received by the traffic light object during the execution of a simple program. These displays are also updated continuously as the program is run.

Finally, Tarraingím allows the user to define abstract views of objects, which can be displayed and maintained by the same framework Tarraingím uses for the low levels views. Figure 3 also shows a simple abstract picture of a traffic light object which is always in accord with the low-level object state displayed alongside it.

Although the above example is very simple, Tarraingím can produce more complex representations for more complex objects, and can produce several views, at differing levels of abstraction, for a single object.

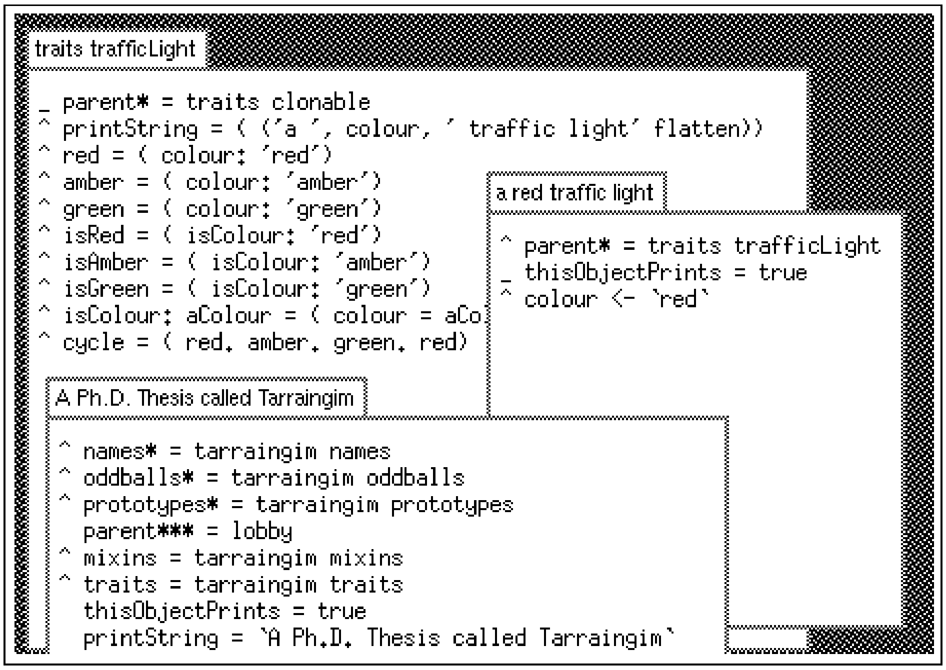

Figure 2: Low level views

4 Design and Implementation

To animate the execution of a computer program, one must first be able to monitor a running program, to provide information upon which to base the displays. This information must then be used to actually draw the visualisations. As Tarraingím is an object-oriented system, it uses specialised types of objects to perform each function. *Encapsulator* objects monitor messages received by particular objects in the target program, while *viewer* objects use information provided by the encapsulators to produce the displays.

4.1 Encapsulators

An encapsulator object [Pascoe, 1986] is created for each object in the target program the user wishes to have animated. All references to the target object in the program are replaced by references to the encapsulator. Any message sent to the target object is now sent to the encapsulator object instead, which forwards it to the target object (thus allowing the program to continue to function normally) but also notifies any dependent objects that the target object has received that message (see Figure 4).

In this way, any number of objects may be informed of the actions of objects in the target program. Note that this also applies to messages passed from an object to itself (so-called "self messages") which SELF uses for slot accesses and assignments.

4.2 Viewers

Viewer objects actually draw the displays of target objects that the user wishes to see. As mentioned above, these illustrations are updated as the program runs, so viewers need to be informed about changes in the state of their target objects. The viewer objects just register their interest in a particular object with that object's encapsulator and then receive the notification automatically.

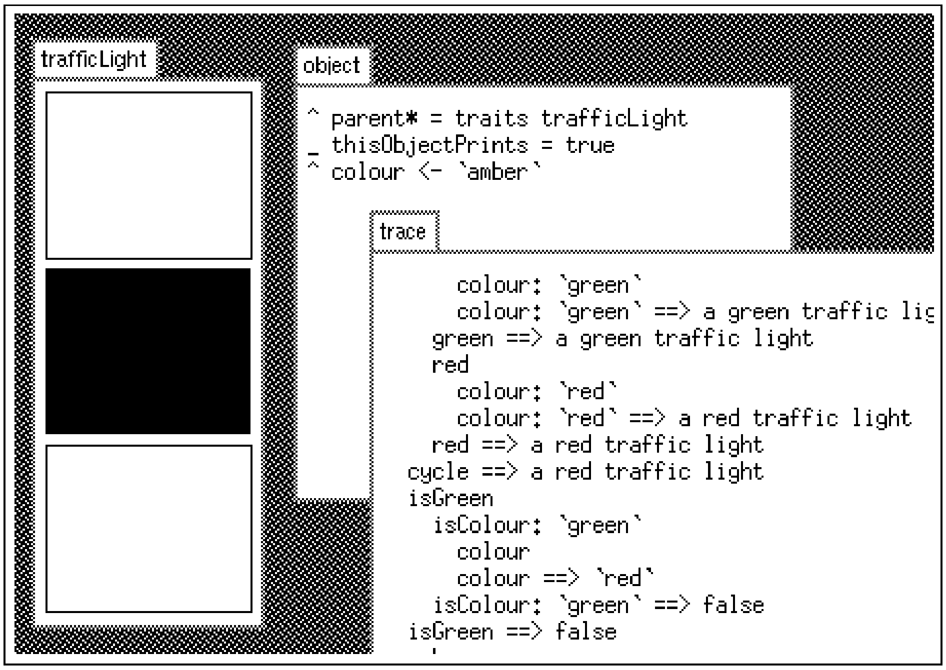

Figure 3: Abstract Traffic Light

There are various kinds of viewer objects. Some are provided by Tarraingím and work for all objects, for example the low level viewers that display an object as a collection of slot and value pairs, or display a textual trace of an object's execution.

Custom viewers (such as the simple traffic light, or bar charts) can also be built to generate an abstract view of any object. These viewers are not provided by the system. They are used to display a picture of an object (or group of objects) which reflects the abstract idea the objects are attempting to model. Precisely because this view is an abstraction, it will not be represented in the source text for any object, and thus cannot be derived from the program source code.

Viewers send messages to the objects they are rendering to collect the information they

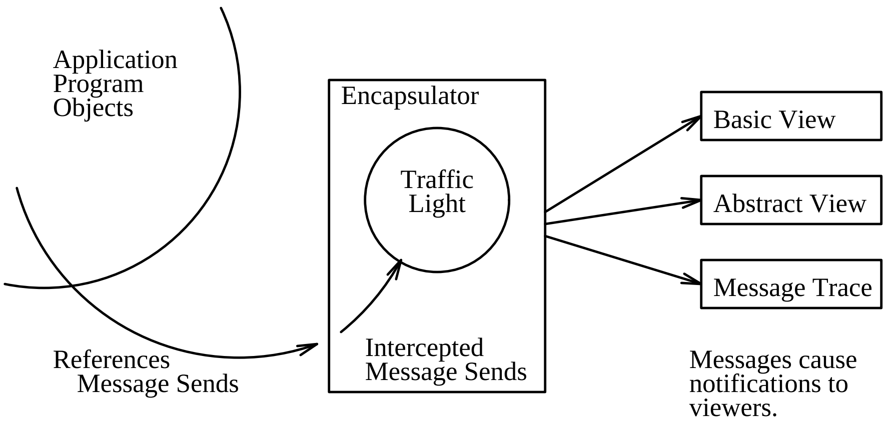

Figure 4: Encapsulator Objects

need to create their views. They do not refer to the concrete internal structure of the viewed objects (except in special cases, such as the very low level views). Thus, they can be used to view any object which represents a given abstraction, by responding to a given message protocol. For example, Tarraingím has a bar-chart viewer, which can portray any SELF "collection" object, including vectors, dictionaries, lists, or strings as a bar chart.

4.3 Indirect Animation

The BALSA system [Brown, 1988] included complex support for "indirect animation" when the displayed picture could not be simply derived from the current state of the animated object. Tarraingím also allows indirect animation, but with a much simpler mechanism. For example, a message profiler can be written by having a "profiler" object receive message notifications (from an encapsulator) and recording frequency counts for the message names in a database object. The database object is itself encapsulated and viewed to produce the profile display, which is updated as the program execution proceeds.

5 Conclusion

The Tarraingím system has been used successfully to animate a variety of small programs, at a couple of levels of abstraction. We are proceeding to test the system with larger and more complex programs. Ultimately, we expect to be able to animate the operation of medium-sized programs, such as a small compiler [Andrews *et al.*, 1988].

5.1 Performance

Some simple measurements have been carried out upon the traffic light example used in this paper. A small program (the "`cycle`" method) was run by a unencapsulated object, and a encapsulated object with various different mixtures of dependent viewers. This test used an otherwise unloaded Sun 3/80 workstation, with 8 megabytes of real memory, and a SELF image using 15 megabytes of virtual memory. The results are shown in Figure 5.

Test		Time (ms)	Factor
(1)	No encapsulator	15	1
(2)	Encapsulator, but no viewers	27	2
(3)	Encapsulator and single viewer	296	20
(4)	Encapsulator and four viewers	13540	900

Figure 5: Encapsulator Performance

These figures suggest that the performance of encapsulators is acceptable, though viewers are rather more expensive. Note, however, that different viewers make different demands upon the system, according to the complexity of the illustrations being produced and the efficiency of their implementation. In particular, the viewer used in test (3) was a very simple one; while one of the viewers used in test (4) was considerable more complex.

In this example, the target program actually does very little; the encapsulator object alone is probably doing more work than the target program. For larger target programs, although there will be more encapsulators, we expect the proportional overhead of the encapsulators to decrease.

We expect that the overhead of actually producing the graphics will always exceed that of the encapsulator mechanism. However, the current viewer objects are quite inefficient, often redrawing the same view of a object several times in succession. We expect to be able to optimise the viewer objects somewhat. However, the overhead of generating the animations will probably remain substantial: the performance of an animated program will always be much slower than the same program without the animation.

5.2 Future Work

Tarraingím needs to be extended in several areas. Currently, there is no user interface to the animation system. The user requests a view by typing a SELF expression at the command line.

There is no support for building user-designed abstract views of objects, other than SELF language access to the X Window System primitives. A graphical design tool would greatly increase the ease of use of the system, and should allow more complex animations to be produced.

As mentioned above, some performance tuning, especially for viewer objects, is necessary. Currently, the viewer objects reside on the same machine, and within the same SELF system, as the target program. It would be possible to separate the viewers from the target program, distributing the animation load over several machines [Brown, 1988].

Acknowledgements

We would like to thank the SELF group at Stanford CIS for their assistance with the SELF implementation. This work is supported by the VUW Internal Grants Committee, the University Grants Committee, and IBM New Zealand.

References

[Andrews *et al.*, 1988] Kristy Andrews, Robert Henry, and Wayne Yamamoto. The design of the UW illustrated compiler. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implmentation*, 1988.

[Brown, 1988] Marc H. Brown. *Algorithm Animation.* ACM Distinguished Dissertation. MIT Press, 1988.

[Chambers *et al.*, 1990] Craig Chambers, David Ungar, and Elgin Lee. An efficient implementation of Self, a dynamically-typed object-oriented language based on prototypes. In Urs Hölzle, editor, *The Self Papers.* The Self Group, 1990.

[Chang and Ungar, 1990] Bay-Wei Chang and David Ungar. Experiencing Self objects: An object-based artificial reality. In Urs Hölzle, editor, *The Self Papers.* The Self Group, 1990.

[Haarslev and Möller, 1990] Volker Haarslev and Ralf Möller. A framework for visualizing object-oriented systems. In *OOPSLA Proceedings*, 1990.

[Henry *et al.*, 1990] Robert R. Henry, Kenneth M. Whaley, and Bruce Forstall. The university of washington illustrating compiler. *ACM SIGPLAN Notices*, 25(6):223–233, June 1990. Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation.

[Hölzle *et al.*, 1991] Urs Hölzle, Bay-Wei Chang, Craig Chambers, and David Ungar. *The Self Manual.* Stanford University, 1.1 edition, January 1991.

[Myers, 1990] Brad A. Myers. Taxonomies of visual programming and program visualization. *Journal of Visual Languages and Computing*, 1(1):97–124, March 1990.

[Pascoe, 1986] G. A. Pascoe. Encapsulators: A new software paradigm in Smalltalk-80. In *OOPSLA Proceedings*, 1986.

[Raeder, 1985] George Raeder. A survey of current graphical programming techniques. *IEEE Computer*, August 1985.

[Stasko, 1989] John Thomas Stasko. *Tango: A Framework and System for Algorithm Animation.* PhD thesis, Brown University, May 1989.

[Ungar and Smith, 1990] David Ungar and Randall B. Smith. SELF: the Power of Simplicity. In Urs Hölzle, editor, *The Self Papers.* The Self Group, 1990.