VICTORIA UNIVERSITY OF WELLINGTON

Te Whare Wananga o te Upoko o te Ika a Maui

School of Mathematical and Computing Sciences

Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 471 5341, Fax: +64 4 495 5045
Email: Tech.Reports@mcs.vuw.ac.nz
Http://www.mcs.vuw.ac.nz/comp/Publications

Rapid Implementation of a Program Visualisation System

John Miller-Williams, Robert Biddle, Ewan Tempero
`http://www.mcs.vuw.ac.nz/Research/renata`

Technical Report CS-TR-98/2
May 22, 1998

Abstract

This paper documents the rapid implementation of a system for visualisation of the execution of C++ programs on a Unix system. The system was to produce dynamic diagrams representing data and control structures so as to assist programmers determine reusable components within the program. Such a system would normally involve a considerable amount of low-level and high-level programming, but we avoided much of thus by extensive use of several programming tools.

We first used the Gnu C++ debugger Gdb, which allows interactive control of C++ program execution, and retrieval of textual information about the program. We then used Expect, a system that allows automatic control of a text interface of a program, to retrieve information about the C++ program using Gdb. In conjunction with Expect, we used the scripting language Tcl and user interface toolkit Tk to analyse the information, draw diagrams, and provide user control.

The end result was a successful program visualisation system produced in a fraction of the time that would otherwise have been necessary. While program visualisation is a relatively specialised application, all this technology would be similarly useful for many other applications.

1 Introduction

Dynamic program visualisation is the application of graphics techniques to produce pictures illustrating the behaviour of running computer programs. Its aim is to display program behaviour in ways that clarify and highlight useful information. The nature of this information varies, depending on the purpose of the visualisation. Dynamic program visualisation can be used to help understand anything from the general principles of computer science down to the workings of an individual program. We are interested in exploring the use of dynamic program visualisation to assist programmers determine reusable components with a program [MW97]. Producing visualisations of programs would normally involve a considerable amount of low-level and high-level programming. This paper describes how, through extensive use of several programming tools, we were able to avoid most of this work in creating a prototype programming visualisation tool.

During the software development process, it is often the case that a large amount of code is rewritten unnecessarily. Research has shown that only a small proportion of code that is written is unique to the individual application [Jon84]. The time and cost of software development could be significantly reduced through better code reuse strategies. One of the main reasons developers do not reuse code to the full potential is the lack of understanding about exactly what reuse is, when it occurs, and how it is done. We have been exploring how tools might help improve such understanding [BT98]. Part of our work has been in determining how dynamic program visualisation can be used to help understand and improve code reuse. We believe the process of constructing visualisations to highlight code reuse will improve our understanding of the topic. Visualising programs that use successful reuse strategies can give insight into better ways to reuse code. The visualisations can also be used to improve programs that do not involve much reuse.

We produced a visualisation tool that displays the dynamic (run-time) aspects of reuse in programs. The visualisation tool is called Dy-re, which stands for Dynamic Reuse profiling tool. Producing Dy-re involved two main tasks: designing the visualisations and developing the architecture for dynamically interacting with the target programs. It is the architecture and its implementation that is the subject of this paper. Many of the design and implementation issues are general in nature, but some relate to specific details involving Dy-re. In particular, Dy-re was written to visualise target programs written in C++, using the Gnu G++ compiler, on a Unix system.

This paper is organised as follows. Section 2 gives a general discussion of Program Visualisation (PV) to provide the context for the architecture of Dy-re. Section 3 then discusses Dy-re's design and implementation. This is followed by a description of Dy-re in action to show the success of the tool. We then present our conclusions.

2 Program Visualisation

PV systems can be classified according to the relationship they have with the program that they visualise (the target program). Target program *dependent* systems display high level views of the execution of a specific program. The visualisation system may have no relationship to the implementation of that program. For example, most algorithm animation systems are target program dependent. These visualise the theoretical aspects of how an algorithm works and are generally used for instructional purposes in the classroom. These visualisation systems are specifically designed to work with a particular algorithm, and cannot be used as general visualisation tools.

Target program *independent* systems display more generic views of the program, generally at the language level. Target program independent systems can be applied to any program

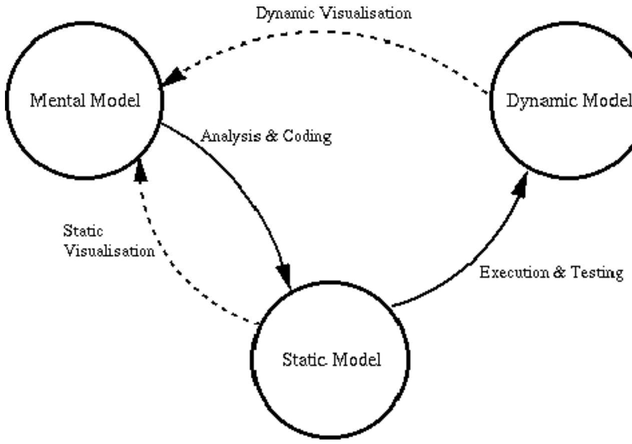

Figure 1: *The mental, static, and dynamic models.*

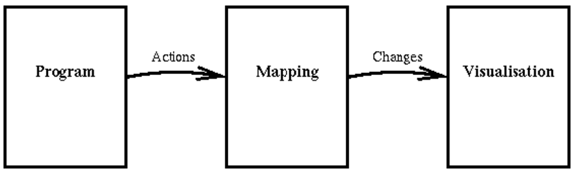

Figure 2: *The PMV model.*

written in a specified language, without the need to set up a special mapping or visualisation routine. For example, an animated interpreter visualises aspects of the interpreter as it interacts with the target program. In general, animated interpreters give highly detailed low-level visualisations of the target program. Another example of target program independent visualisation systems are graphical debuggers. Graphical debuggers visualise the basic behaviour of the target program, such as the control flow and the data structures. Dy-re will be a target independent visualisation tool that can visualise any program written in a specified language, C++.

Program visualisation can also be classified as static or dynamic. To help explain this difference, we examine a model of the process of software development [WO95]. This model is illustrated in figure 1. Software development begins with the programmer forming a mental model of the program in their head. The mental model is then transformed into a static model using analysis, design, and coding. This is done using development tools such as text editors, CASE tools, and compilers. The result is the source and executable code.

Static program visualisation assists transforming the static model back to the mental model. It displays unchanging features of the program structure that can be determined directly from the source code. The static model is then transformed into the dynamic model via execution and testing. The dynamic model reflects the run-time behaviour of the program. The dynamic model can be mapped back to the mental model using dynamic visualisation.

Dynamic visualisation displays information about the operational nature of the program, such as changing relationships, and program interaction. Static visualisation is important, but it is not enough to gain a full understanding of a program. Dynamic execution can be difficult to predict, and can differ greatly from the static model.

The PMV model, created by Stasko and Roman and Cox, describes an architecture for program visualisation [RC93]. We have been influenced by the version developed by Noble [Nob95]. Figure 2 illustrates this model. It is based on the concept of a mapping from the target program to a resulting visualisation. The PMV model is named after its three components: the program component, the mapping component, and the visualisation component. Each component represents a transformation along the mapping process. The following sections discuss each component in more detail.

2.1 Program Component

The program component interacts with the target program. Its purpose is to determine what the target program is doing, and send information, in the form of *actions*, about what has occurred on to the mapping component. This information can vary. For example, an action might be a function call or an object instantiation. Actions can contain extra information about the target program, such as information about the performance of the program.

Communication between the program and mapping components can take a variety of forms. For example, the information can be saved to a log file as the program runs, and then be visualised at a later date. The most common option is for the program to be temporally suspended at points while the PV system updates the display. If a parallel system is available, the target program and the PV system can be run simultaneously.

The method of obtaining the actions from the target program also varies widely. Action generation can come from three possible phases in program creation: at the source code phase, the compiler phase, and at the executable phase.

2.1.1 Source Code Phase

The source code can be modified to generate the actions. The most straightforward way to do this is for the user to annotate the source code. The user inserts calls to the visualisation routines in the source code so that when the program is run, these calls create the events to drive the visualisations. The problem with this method is that the source code must be edited every time a change is required of the visualisation. A better method is to use a pre-processor to automatically annotate the source code. The difficulty in this approach is that the pre-processor must be able to interpret the source code sufficiently to perform the task.

2.1.2 Compiler Phase

The compiler can be modified to generate the actions. By modifying the language processor, the compiler semantics are changed to include the event generation routines. This is transparent to the user, but portability is reduced, as the language processor must be re-implemented for each different platform. Some programming languages have the property of being reflexive. This means parts of the programming semantics are written in the language itself. This enables the language to change its own computation at the base level. Therefore the language itself can be modified to include event generation routines. This is a highly transparent way of creating events, but requires the target program to be written in the reflexive language.

2.1.3 Executable Phase

Finally, actions can be generated directly from the executable. Post-processing the executable uses information contained in the compiled binary to generate events. This method is highly transparent; neither the source code nor the compiler needs to be modified, however only information that is contained in the executable can be used. Another possibility is hardware monitors that look at the low level hardware events created by the program as it executes. The low-level operations are then abstracted to higher level events that are used to drive the visualisations. The advantage of this approach is that it is highly transparent to the user. The disadvantage is that the events are generally very low level, and need complicated processing to be interpreted.

2.2 Mapping Component

The mapping component translates the actions sent by the program component into a format, called *changes*, that can be used to update the visualisations. For example, an action would be a function being called, while the change would be a request to display this event in the visualisation. This means the visualisations no longer have a direct dependency on the low-level output of the program component. Additionally, the mapping component might perform the following tasks:

- The mapping component can focus on desired information and filter out unwanted information. A program execution involves a large number of instructions and amount of changing data. It is unlikely that the user would want to see, or be able to understand, all this information.

- The mapping component can summarise data and output abstract summaries to reduce complexity. For example, a sequence of actions might represent an abstract meaning that can be sent as a single change. Information from a large number of events can be sampled and the summary sent to the visualisations.

- Abstract visualisations require changes that are meaningful to the existing abstractions, rather than the direct changes of the target program. Abstraction recovery is the process of mapping the actions produced by the program component into changes that the visualisation component can understand.

2.3 Visualisation Component

The visualisation component receives the changes from the mapping component and uses these to produce and update the output images. The operations of the visualisation component can also vary widely. Multiple visualisations can be displayed simultaneously. The output can be produced once, or updated during the execution of the program to create an animated display. Visualisations can be interactive or non-interactive. Interactive visualisations allow the user to modify the graphical features of the display. For example, the user could set display colours, arrange the layout, or focus the visualisation on specific areas. At the most advanced level the user can fully define the visualisations using a specification language. Non-interactive visualisations are set and have fixed displays. Most PV tools have specific purposes and therefore dedicated visualisations providing only basic interaction.

3 Design and Implementation

In the section above, we have discussed general issues regarding program visualisation. Given our motivation for building Dy-re, it was clear that it had to be a target independent, dynamic visualisation system. It had to be the case that it could deal with any program written in a specified language, C++, without the need to set up a special mapping from the target program. In this section, we describe the design and implementation of Dy-re. In particular, we outline how we were able to implement Dy-re very quickly by harnessing the power of other tools.

The architecture of Dy-re is based on the PMV model discussed in the previous section. This model provides us with a way of dividing the visualisation system into manageable

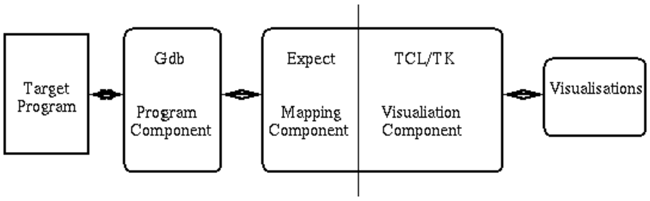

Figure 3: *The structural divisions of the Dy-re.*

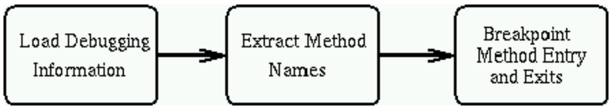

Figure 4: *Loading a target program.*

components. The architecture of the system is illustrated in figure 3. The system is composed of three logical sub-systems that correspond to the three components of the PMV model. A pure representation of the PMV model might separate the components completely and develop a communication protocol between them. However, in our case, it proved easier for the mapping component and the visualisation component to be parts of the same physical program.

We now detail the design and implementation decisions of the three components.

3.1 The Program Component

The program component interacts with the target program. It extracts run-time events from the target program, and passes these actions on to the mapping component. Dy-re's program component is Gdb, the Gnu source level debugger. Gdb is a debugger for C and C++, with a rich text command interface, and good capabilities for breakpointing and data access.

The decision to use Gdb was a central factor in the rapid development of Dy-re. Using Gdb allows easy access to the information that Gdb can extract from the target program, using debugging information that has been inserted into the executable code by the compiler. Recalling our discussion in the previous section, using Gdb means we have chosen to address the PMV program component in the "executable phase". Gdb also provides the ability to determine information about the program, run the program, have it stop on various conditions, and allow access to the current state of the program, such as the call stack.

Figure 4 illustrates how Gdb is used to process a target program. When the user selects the target program, Gdb reads in the debugging information contained within the executable. Gdb extracts the names of all the methods in the target program (excluding the basic system library routines). It then places breakpoints at the start and end of these methods so that the program will halt at these places. Dy-re makes use of the following Gdb commands to perform this:

- `file` *program name* — Sets the target program to be debugged. The debugging information in the target program is read, and it is the program executed when the run command is issued.

- `info types` — Lists all the names of the types contained in the target program. We use this command to find all the class names.

- `rbreak` *regular expression* — Sets a breakpoint on all functions matching the regular expression. We use this function to set breakpoints on the methods of all the classes.

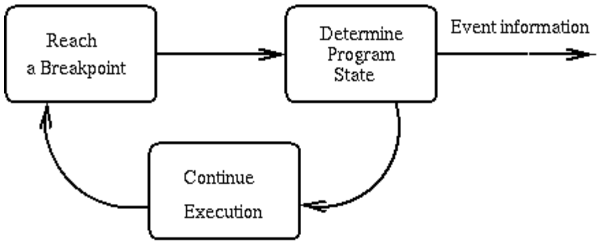

Figure 5: *Running a target program.*

Figure 5 illustrates how Gdb executes the target program. When the user runs the program, if a message call or return occurs, the program will halt. At this point it is possible to extract information about the current state of the program (for example, the call stack). The program is then restarted, so that from the user perspective, the visualisations appear to be updated continuously. Dy-re uses the following Gdb commands to do this:

- `run` — Starts the target program executing.
- `backtrace` — Displays a trace of the run time stack.
- `info frame` — Displays further information about a particular stack frame.
- `continue` — Continues the target program executing after a breakpoint.

3.2 The Mapping Component

The mapping component translates the low-level actions sent by the program component into changes that can be used to update the visualisations. When designing the mapping component, the following implementation features needed to be considered: firstly, how should the mapping component interact with Gdb? Secondly, as Gdb controls the target program, how can the end user interact with the program directly? Finally, as the visualisations must provide abstraction (information hiding), what data structures are required to keep an internal representation of the program execution?

The first implementation problem was how the mapping component should interface with Gdb. There were two options: the first was to "hard-wire" the mapping component to the internal parts of Gdb to directly access data and control the program. The other was to control Gdb externally through the text-based interface, sending in commands and parsing the output.

Modifying Gdb's internals would remove a layer of communication and mean faster access to program information. Communicating externally with Gdb, and parsing the output of the program, would improve portability. Changing Gdb's source code would mean that the modified version would need to be kept with the visualisation system. Gdb is a reasonably large program, and most Unix systems already have a version installed. Keeping Gdb as a separate entity results in a small compact tool that can be easily transported.

The decision to communicate externally with Gdb was mainly due to the availability of Expect [Lib95]. Expect is designed to automate interactive text based programs. It communicates with these programs using a script written by the user. Following the script, Expect knows what output is expected from a program and what the correct response should be. As the user manual says, "All that is necessary is that the interaction can be characterised programmatically". The output of Gdb is fairly formalised and therefore not particularly difficult to parse using Expect. Moreover, we discovered that communicating externally did

```
#
# Initialize the connection to GDB.
#
proc ogdb_init {} {
    ...
    spawn -noecho gdb  --quiet --nx --tty=/dev/[lindex $argv 0]
    ...
#
# Load the details of $file into GDB.
#
proc ogdb_open {file} {
    ...
    exp_send "cd [exec pwd]\n"
    exp_send "file $file\n"
    ...
#
# Determine the list of classes using GDB's "info types" command.
#
proc get_types {types_name} {
    ...
    exp_send "info types\n"
    ...
    expect {
        -re "\nFile (\[^\n\]*):\r" {
            set file $expect_out(1,string)
            set types($file) {}
            exp_continue
        }
        -re "\n(\[^ ;\n\]+);\r" {
            set type $expect_out(1,string)
            exp_continue
        }
    }
    ...
#
# Set breakpoints for each of the class methods using GDB's "rbreak" command
#
proc break_methods {type} {
    ...
    exp_send "rbreak ^$type::\n"
    expect {
        -re "\nBreakpoint (\[0-9\]+) at (\[^:\]*):
             file (\[^,\]*), line (\[0-9\]+)\.\r\n(\[^;\]*);\r" {
            set breakpoint $expect_out(1,string)
            set address $expect_out(2,string)
            set file $expect_out(3,string)
            set line $expect_out(4,string)
            set name $expect_out(5,string)

            set break($breakpoint,address) $address
            set break($breakpoint,file) $file
            set break($breakpoint,line) $line
            set break($breakpoint,type) method_call
            ...
```

Figure 6: *Excerpts from the Expect code using Gdb to load the target program, find classes, and initialise breakpoints.*

```
#
# Run the client program through GDB, generating an event every breakpoint.
#
proc ogdb_run {{tag default}} {
    ...
    exp_send "run\n"
    expect {
        -re "\nBreakpoint \(\[0-9\]+\), \(.*\) at \(\[^:\]+\):\(\[0-9\]+\)\r" {
            expect
            set breakpoint $expect_out(1,string)
            set info $expect_out(2,string)
            set file $expect_out(3,string)
            set line $expect_out(4,string)

            switch $break($breakpoint,type) {
                method_return {
                    exp_send "delete breakpoints $breakpoint\n"
                    expect
                    set trace $break($breakpoint,info)
                    trigger_event $tag return $trace
                }
                method_call {
                    get_stack stack $ogdb_maxdepth
                    set trace {}
                    for {set f 0} {$f < $stack(size)} {incr f} {
                        lappend trace $stack($f,location)
                    }
                    if {$stack(size) > 1} {
                        set b [set_tbreak "*$stack(1,address)" method_return]
                        set break($b,info) $trace
                    }
                    trigger_event $tag call $trace
            ...
#
# Determine a complete stack trace using GDB's "info frame" command
#
proc get_stack {stack_name max_depth} {
    ...
    exp_send "backtrace\n"
    ...
    for {set f 0} {$f < $max_depth} {incr f} {
        exp_send "info frame $f\n"

        set break 0; set frame ""
        expect {
            -re "\n(Stack frame at $re_address:\[^\r\]*)\r" {
                set frame "$expect_out(1,string)\n"
                exp_continue
            }
            ...
```

Figure 7: *Excerpts from the Expect code using Gdb to run the target program and handle breakpoints.*

not significantly reduce efficiency. The primary delays experienced when using the program appears to caused by Gdb itself.

Figures 6 and 7 show excerpts from the Expect code that uses Gdb to extract information from the target program, control execution of the target program, and report the details to other parts of Dy-re.

The second implementation problem was the requirement that the user must be able to interact with the target program directly. As the target program was run indirectly through Gdb, it meant the user had to wade through the event extraction process. However, Gdb can redirect the debugging information and the target program output to different places. Therefore, Gdb's input/output was redirected to an internal output that could be processed by the mapping component, and the target program input/output was redirected to the shell window. Figures 8 and 9 show an example program using its usual text interface, while Dy-re presents its own interface to control the visualisation diagrams.

The final implementation problem involved creating data structures to keep an internal representation of the state of the program. Because we intend to focus on how program units are used and reused, we need to keep track of *dependencies* which show how units of the program are used together. The following information needed to be stored:

1. Dependency history. As sub-sections of the program trace may not be shown all the time, it is necessary to keep an internal representation of the history of call dependencies.

2. Dependency existence. The class visualisation displays the existence of dependencies, so we will need to record any dependency as it happens.

For efficiency, we keep an internal representation about each dependency, rather than computing it from the history data. It is also necessary to store additional information about the dependencies. The frequency of the occurrence of the dependency needs to be stored. The system must also recognise occurrences of polymorphism. Gdb does not recognise polymorphism itself, but it does provide the location that a method is called from. Our solution is to record, with each call dependency, the location of the call and name of the method called. If the same location makes calls to different methods, then it implies an occurrence of polymorphism.

3.3 The Visualisation Component

The visualisation component takes incoming actions from the mapping component and uses them to update the various views. This component is composed of two parts: the visualisations themselves and the supporting structure for the visualisations. The supporting structure provides routines to activate and destroy the visualisations, and acts as a controller ensuring each visualisation gets only relevant updates.

The visualisation component is also responsible for providing the user interface. There are two types of interface. Firstly, there is the direct manipulation of the visualisations. Each visualisation is responsible for this. The visualisations provide event driven control of the objects they display. Users can manipulate the diagrams to select the required information and to clarify the presentation. Secondly, there is the control panel interface that allows the user to control the target program and decide on the visualisations to view. The main control is based on a tape deck metaphor (see figure 9). This gives the user an intuitive way to manipulate the program events. The following section shows Dy-re in action, and the figures show both the user interface and visualisation diagrams.

In Dy-re we decided that the visualisation component should be implemented using Tcl, and the Tk graphics library [Ousterhout, 1994]. Tcl/Tk already provides a number of quite

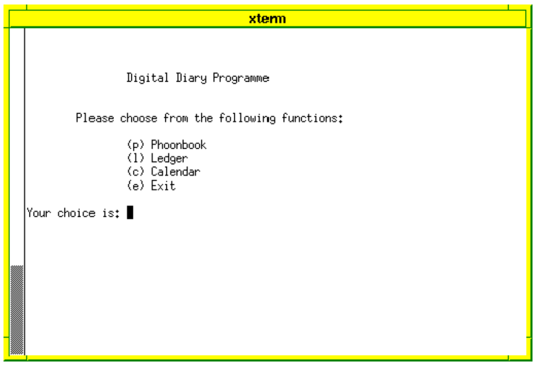

Figure 8: *Interacting with the target program: its usual interface is unaffected by the visualisation system. In these examples, the target program is a text-based digital diary program.*

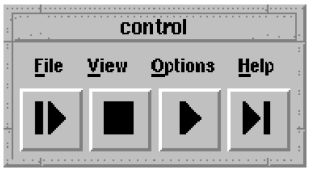

Figure 9: *The main interface for Dy-re itself.*

high-level graphics functions for manipulating the provided library of graphical "widgets". Furthermore, Expect works by using Tcl, and use of Tcl/Tk for the visualisations means that connection between the mapping and visualisation components is easy. In fact, the version of Expect we use is "Expectk" [Lib94], a combination of Expect and the Tk toolkit. As a result, it was possible for the mapping component and the visualisation component to be two parts of the same program. This then became our design, and communication between the mapping and visualisation components was made even easier because the mapping component could create data structures which were usable by the visualisation component.

4 The Visualisation System in Action

In the section above, we described the design and implementation of Dy-re, explaining how development was made rapid by the use of Gdb, Expect, and TCL/Tk. While the rapid development was interesting in itself, it must really be considered in the context of the results. In this section we therefore show Dy-re in action.

Our primary work is in software reuse, and Dy-re is a prototype tool for research to help programmers better understand features of programs that relate to reuse and reusability. In this particular project, we were concerned with C++ programs. C++ supports a design emphasis on static structure, so tool support based on the static structure can help a great deal. But, no matter how static the behaviour of a program, the static structure can never tell the whole story. The static structure of a program can really only outline possibilities: what *might happen* when a program is executed. The dynamic structure is what actually *does happen* at execution time. Illustrating these dynamic aspects of reuse was the main job of Dy-re.

Dy-re runs the target program, and the program may be paused, resumed, or restarted at any time. As the program runs, Dy-re displays diagrams. The diagrams are inspired primarily by our more general research into programming language support for reusability [BT96], where we discuss in detail various program units we call *assemblies*, and the *dependencies*

Figure 10: *An object with a method call to another object completed, and return underway (with the envelope stamped "R").*

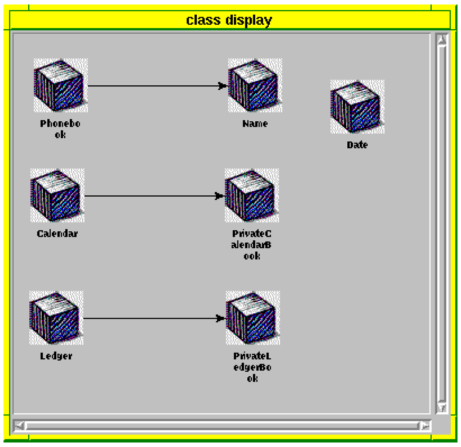

Figure 11: *Initial classes and dependencies: this diagram represents the state at the beginning of program execution.*

between them. In particular, the diagrams show invocations from *context* assemblies to *component* assemblies, and attempt to highlight both component reuse and context reuse. The Dy-re diagrams are also influenced by other research work in program visualisation involving dynamic factors, such as that of Jerding [JSB96] and that using in the Look C++ debugger [And95].

In the Dy-re diagrams the assemblies are typically objects (instances of classes) and the dynamic structure of the program is shown where different objects are involved in different dynamic structures. The diagrams show objects as they are created. When one object calls a method of another object, a line is drawn between the objects, and an envelope is animated moving in one direction at the call, and back when the call method returns (see figure 10). Any call still current is represented by a red line, and calls already complete are represented by black lines. The thickness of the line is increased as calls from distinct places are detected, to greater emphasize the occurrence of reuse, and highlight potential usefulness of particular program units. Figure 11 shows the program soon after it begins, and figure 12 shows the situation further into execution.

Objects can be expanded, as illustrated in figure 13, thus showing the methods of the object. When calls are made to or from expanded objects, the lines connect to the methods involved. Expanded objects are displayed transparently against the background diagram to aid comprehension that diagram reorganisation might otherwise hinder. These animated diagrams make it clear which objects use and are used by which other objects, and also extend to the methods within objects. This emphasis allows us to show concepts clearly, and supports understanding of how both context and component reuse work at run time.

To try and support understanding of larger scale dynamic patterns, the tool also provides a "trace view" which shows how objects use each other over time, as shown in figure 14. The trace view is based on the run-time call stack, and the calls from one object's method to another object's method are depicted from left to right on the horizontal axis. When a

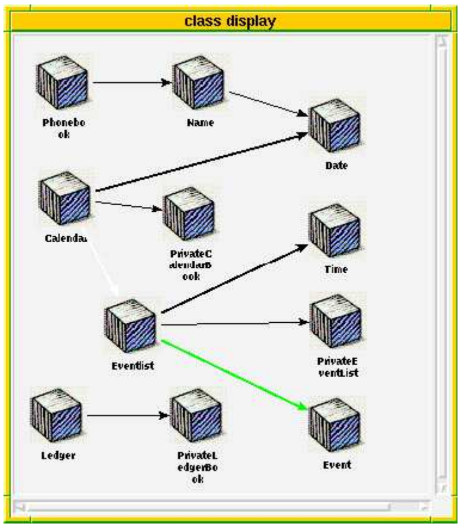

Figure 12: *Midway through execution, several other objects have been created and are in use.*

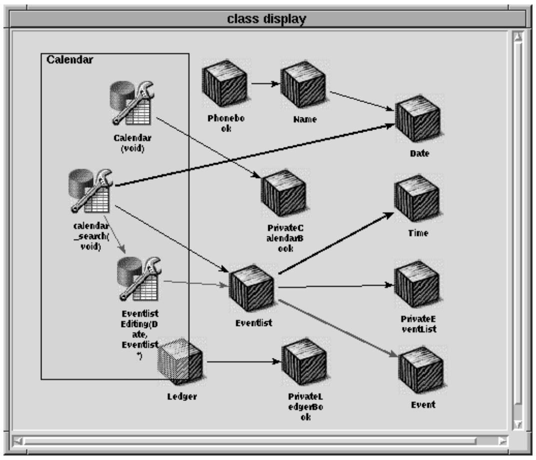

Figure 13: *An expanded object, showing which methods within the object have made calls to other objects.*

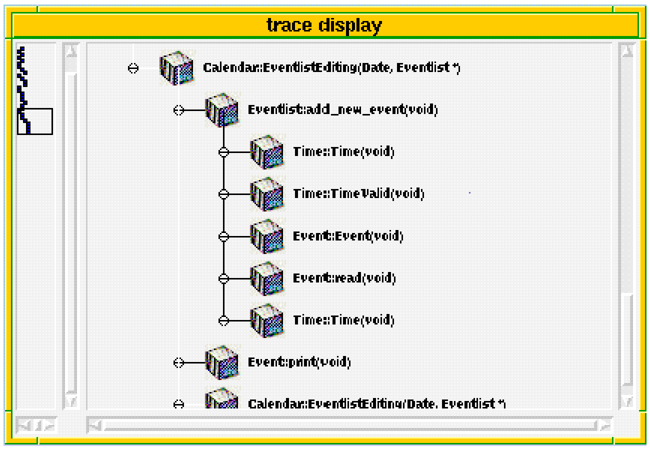

Figure 14: *The trace diagram, showing the history of the run-time stack. The window may be scrolled, and the panel at the left shows a global view in lower resolution, with the currently visible section indicated by the rectangle.*

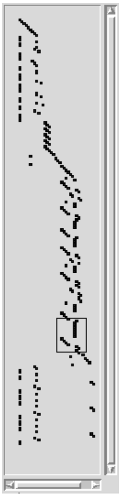

Figure 15: *The global view displayed as a diagram by itself, showing how message patterns can be detected.*

method returns it remains shown, and later calls from the same method are then displayed below using the vertical axis. In this way, any period of execution is shown as a sideways tree, with the root at the left, and any path through the tree represents the run-time stack at a particular time. Together the horizontal and vertical axes are used to show the run-time trace over a period of time. The trace view is displayed in real time as the program executes.

The trace view allows the user to see repeated message patterns in the dynamic structure, as structural repetition is depicted as graphical repetition. To make it easy to see such patterns over longer periods of time, a "global view" is also provided, as shown on the left of figure 14. The smaller size used for methods allows for less detail, but for longer periods to be visible, so enabling detection of longer-term patterns. The global view can be shown as a diagram itself, and can be adjusted to show methods in individual columns, further facilitating pattern recognition (see figure 15).

5 Conclusions

In this paper, we presented the design and implementation of Dy-re, a program visualisation tool to help us determine how dynamic program visualisation can be used to help understand and improve code reuse.

The main lesson learned from the implementation of Dy-re concerned the value of scripting and the Gnu source level debugger for C and C++, Gdb. We used Gdb to give us interactive control of C++ program execution, and retrieval of textual information about the program. We then used Expect, a system that allows automatic control of a text interface of a program, to retrieve information about the C++ program using Gdb. In conjunction with Expect, we used the scripting language Tcl and user interface toolkit Tk to analyse the information, draw diagrams, and provide user control. This method proved easier, more flexible, and more powerful than we expected. As a result, we now view dynamic analysis as less forbidding, and are exploring new ideas that might use this approach. While program visualisation is a relatively specialised application, all this technology might also be similarly useful for many other applications.

References

[And95] Eddie Anderson. Dynamic visualisation of object programs written in C++. Objective Software Technology Ltd, Software Innovation Centre, 1 Michealson Square, Kirkton Campus, Livingston, Scotland, EH54 6QA, 1995.

[BT96] Robert Biddle and Ewan Tempero. Understanding the impact of language features on reusability. In Murali Sitaraman, editor, *Proceedings of the Fourth International Conference on Software Reuse*. IEEE Computer Society Press, April 1996. Also available as Technical Report CS-TR-95/17.

[BT98] Robert Biddle and Ewan Tempero. Towards tool support for reuse. In *Proceedings of the Conference on Software Engineering: Education and Practice*, Dunedin, New Zealand, January 1998. IEEE Computer Society Press.

[Jon84] T. Capers Jones. Reusability in programming: A survey of the state of the art. *IEEE Transactions on Software Engineering*, SE-10(5):488–494, September 1984.

[JSB96] Dean F. Jerding, John T. Stasko, and Thomas Ball. Visualising message patterns in object-oriented program executions. Technical Report GIT-GVU-96-15, Graphics, Visualisation, and Usability Center. Georgia Institute of Technology, May 1996.

[Lib94] Don Libes. X wrappers for non-graphic interactive programs. In *Proceedings of Xhibition '94*, San Jose, CA, June 1994.

[Lib95] Don Libes. *Exploring Expect: A Tcl-based toolkit for automating interactive programs.* O'Reilly and Associates, 1995.

[MW97] John Miller-Williams. A program visualisation tool for emphasising the dynamic nature of reusability. Master's thesis, Victoria University of Wellington, 1997.

[Nob95] Robert James Noble. *Abstract Program Visualisation: Object Orientation in the Tarraingim Program Exploratorium.* PhD thesis, Victoria University of Wellington, 1995.

[RC93] Gruia-Catlin Roman and Kenneth C. Cox. A taxonomy of program visualisation systems. *IEEE Computer*, 26(12), December 1993.

[WO95] Hendra Widjaja and Michael J. Oudshoorn. Devising a program visualisation tool for visualisation of concurrent and object-oriented programs: A survey. Technical Report TR95-14, Department of Computer Science, University of Adelaide, Australia, December 1995.