BALLARAT UNIVERSITY

ABSTRACT

A PROGRAM VISUALISATION META LANGUAGE

by David Stratton

A program visualisation system sets out to provide visual representation of the execution of a target program in the hope that this will help programmers better understand the effect of the program code. Despite the intuitive appeal of this technique there is still a lack of conclusive, empirical evidence that supports its efficacy. Experimentation in this regard has been conducted using a variety of visualisation systems each of which incorporates a particular approach to visual representation and usually a particular programming language. There has been little opportunity for educational and psychological researchers to test the effect of varying these approaches and this limitation arises from the monolithic nature of most program visualisation systems. The proposed Program Visualisation Meta Language provides a generalised communication between an arbitrary executing target program and an engine that provides visual representations of execution. This decoupling of target and engine offers an increased scope for experimentation in the field.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

The part-time doctoral degree is a magnificent concept, from the point of view of a university teacher who is significantly committed to a professional lecturing career, yet seeks further academic advancement. It is very far from a magnificent concept for the wife of that teacher. My wife, Ros, has had to contend with six years of a distracted, and increasingly frantic, husband and her support, and love, have been central to this endeavour.

My principal supervisor, Dr Philip Smith, has been a fellow traveller, and has shown tireless dedication to Program Visualisation, his own doctoral topic, and the potential impact of this research on that domain. My warm thanks go to Philip and also to Dr John Wharington, my associate supervisor, whose technical insight into the programming domain, and willingness, literally, to travel the extra mile in supervising me, has been a significant contribution.

Lastly I wish to acknowledge Professor Sidney Morris, my head of school, who bent over backwards to relieve me of teaching duties in the final phase of my completion.

ABBREVIATIONS AND ACRONYMS

ADT Abstract Data Type

AI Artificial Intelligence

ANF Attribute Normal Form

API Application Programming Interface

ASCII American Standard Code for Information Interchange

AV Algorithm Visualisation

BNF Backus-Naur Form

CASE Computer Aided Software Engineering

CLR Common Language Runtime

COM+ Component Object Model

CORBA Common Object request Broker Architecture

CS1, 2 Computing Science Year 1 (2)

DOM Document Object Model

DTD Document Type Definition

ENF Element Normal Form

GCC GNU Compiler Collection

GDB GNU Debugger

GDL General purpose Debugging Language

GNU GNU's Not Unix (!)

GUI Graphical User Interface

HTML Hyper Text Markup Language

HTTP Hyper Text Transfer Protocol

IDL Interface Definition Language

IP Internet Protocol

JDB Java Debugger

JDI Java Debug Interface

JDK Java Development Kit

JPDA Java Platform Debugging Architecture

JTB Java Tree Builder

MIME Multi-purpose Internet Mail Extension

PE Programming Environment

PV Program Visualisation

PVML Program Visualisation Meta Language

RFC Request For Comment

RMI Remote Method Invocation

RPC Remote Procedure Call

TCP Transmission Control Protocol

SAX Simple API for XML

SV Software Visualisation

W3C World Wide Web Consortium

XML Extensible Markup Language

Chapter 1

INTRODUCTION

This research is concerned with the difficulties that arise when people learn to program computers. A novice programmer is confronted by a daunting learning curve, part of which is the need to acquire mental models of the process. It has been suggested by many authors, that the mental models of the novice might be aided by systems that provided visual representations of the program they are writing, in order to reinforce the largely text-based view that is prevalent. This approach, Program Visualisation (PV), has an intuitive appeal but its efficacy has not been conclusively and empirically demonstrated.

Research into the efficacy of PV is, not necessarily, a computer science undertaking. Significant contributions could potentially be made by educational and psychological researchers but PV systems are not generally open to reconfiguration by non computer scientists. A visualisation system generally incorporates a particular approach to visual representation and usually a particular programming language. The opportunity for educational and psychological researchers to test the effect of varying these approaches is limited and this limitation arises from the monolithic nature of most program visualisation systems.

The proposed Program Visualisation Meta Language provides a generalised communication between an arbitrary executing target program and an engine that provides visual representations of execution. This decoupling of target and engine offers an increased scope for experimentation in the field.

This thesis is submitted as the major component of the research portfolio for this professional doctorate.

1.1 Background

Computer programmers are often puzzled by the effects of the program that they have written. For the novice programmer the problem is compounded by the fact that they usually have inadequate mental models of the entire programming process. To examine the behaviour of a running program expert programmers have historically resorted to adding lines that print messages or values to the screen. More sophisticated programmers might use a debugger to step through their program and inspect its behaviour. Neither of these approaches offers much help to the struggling novice whose lack of understanding of the programming process can often leave them confused and demoralised.

An alternative, one that might seem especially attractive to novice programmers, is to provide some means of offering a more tangible representation of program execution. The hope is that pictures or sounds representing the state of the program will assist the development of mental models of the execution process. The domain of PV has been the location of much research, development and effort within computing science and many large and complex systems have been created to provide, mainly visual, representations of program execution across a broad range of computer languages.

Naturally enough, the development of PV systems has been accompanied by research into their efficacy, largely focusing on the question of whether novice programmers are significantly assisted by the use of PV. Typically the developers of a PV system, usually university researchers, will survey the students who have used their system. In some cases they might conduct experiments in which new programming students will be exposed to programming pedagogy both with, and without, the PV system. Although the qualitative studies have generally favoured PV the somewhat surprising

conclusion of the quantitative work is that it has yet to be shown, convincingly, that novice programmers benefit from PV.

It is against this background that the author has formed an interest in the pedagogy of computer programming. In [106] the author has proposed a "location and programming language independent" novice programming environment. An argument has been presented for the provision of a programming environment in which the target program, the one that is being written and tested, is at a location that is remote from the novice programmer. It has also been suggested that such an environment might incorporate PV features. It is the proposal to provide PV in an environment that is distributed, and which sets out to support programming in a variety of languages, that led to the initial formulation of the Program Visualisation Meta Language (PVML) proposal.

1.2 Motivation

The background described, both in terms of the author's suggested novice programming environment and the significant uncertainties surrounding the usefulness of PV for novice programmers, together provide the motivation for this research. In particular the motivation with regard to the general area of PV research, is worthy of further explanation in this thesis. The more general issue of a novice environment is covered in Chapter 2.

The question of what is, or is not, pedagogically effective is one that is generally addressed by researchers in the field of education and psychology. These researchers have learned to apply a range of statistical and experimental techniques and are conversant with the psychology of perception and the development process of mental models that students undergo. Despite this the bulk of research into PV has been conducted by computer scientists, the designers and builders of the PV systems. In general the field of PV research has not been accessible to more educationally oriented researchers. The closed nature of PV research relates directly to the

closed nature of PV systems. A given PV system provides a particular visual representation of execution for a particular programming language yet empirical research in the field would perhaps seek to compare a variety of visual approaches to pedagogy amongst a cross-section of computer languages.

A careful review of PV literature reveals that comparatively few researchers have explicitly isolated the role of *visualiser*. It is the role of a visualiser to make the potentially pedagogically significant decisions as to what form of visual representation will be used to represent particular programmatic artifacts and states. This role, most often implicitly filled by the designer of the PV system, is the location for precisely the pedagogical decisions that should be examined most closely. Again it is the monolithic design of most PV systems that fails to provide satisfactory access for the visualiser role.

The PVML proposal has the potential to decouple, or componentise, PV systems; introducing a strict boundary between the executing program, which is termed here the *target*, and the means of providing visual representation, which is referred to as the *engine*. By establishing this boundary, across which only program state information flows, it is possible for arbitrary engines to communicate with arbitrary targets. A particular visualisation approach, represented by a PVML engine, can therefore be applied to targets incorporating a variety of programming languages. Alternatively novices, learning a particular programming language, can apply various visualisation engines, employing different visual metaphors, to the task of understanding their particular program.

The effect of this should be to define a generic location for the activities of the visualiser and hence to expose PV research as an area for educational rather than computer specialists. Visualisation engines, that incorporate explicit visualiser tools and interfaces, can expect to communicate via PVML with a wide range of targets. The generalisation of this access implies that

the effort expended in generating new and more sophisticated visualisation tools can expect wider access and larger markets than would otherwise be expected.

The principle motivation of this work is therefore to define an open PV architecture that will enable a variety of visualisation schemes to interoperate and that will encourage the generation of PV systems and research into their efficacy. Ultimately this may lead to more effective pedagogy in the field of computer programming and hence remove a barrier to students entering the profession.

Computer programs, their creation and maintenance, occupy a critical position in the twenty-first century economy. Programming related endeavours represent a substantial element within that economy, but one that is constrained by the supply of competent and well trained computer programming professionals. Helping the novice programmer in their struggle to engage with the field is a first step to securing that supply.

1.3 Contribution

The effect of a convincing definition of a Program Visualisation Meta Language will be to open the PV field to significant innovation.

On the one hand programming languages that are used pedagogically, but for which no visualisation tools are available, can potentially be visualised by a range of PVML compliant visualisation engines. Providing such additional targets involves wrapping a debugger for the language with appropriate PVML drivers. If it is assumed that PV is useful for novice programmers, the approach becomes accessible to those learning a greater cross-section of languages.

On the other hand diverse approaches to visualisation can be implemented in PVML compliant engines. In particular attention can be paid to

configuring such engines in a manner that supports a meaningful visualiser role in order that non computer scientists can configure, evaluate and assess varied approaches to visualisation. This has the potential, perhaps, to lead to some better answers to the question "Does PV help novice programmers?"

Although PVML has been characterised as a development that will encourage further research in the field, the potential encouragement that the decoupled architecture provides for PV software development should not be neglected. It has been argued that the effect of componentisation in other software development fields has been to encourage the growth of those fields. PVML represents a critical step towards the componentisation of PV systems and as such, a significant contribution to their future proliferation and development.

1.4 Overview

The description of the proposed Program Visualisation Meta Language is supported by three chapters that assess related work in the field.

Chapter 2 examines novice programming environments in general and sets out to underpin the proposition that a location and language independent novice programming environment would be pedagogically useful. This represents the earlier stages of the study undertaken in this doctorate, and concludes by suggesting that the provision of PV features within such an environment would represent a significant challenge.

Chapter 3 specifically addresses the field of program visualisation and examines the various approaches taken in the history of this field. PV is examined from various angles and special attention is paid to work that has set out to define taxonomies of PV systems.

Chapter 4 addresses the predominant issue in program visualisation from the point of view of the PVML proposal – the decoupling of visualisation targets

and engines. The PV systems examined here are those which partition the PV problem along similar lines to that adopted by PVML.

Chapter 5 begins with a very broad, architectural, definition of PVML and moves on to locate the language in the field of debugging. Given that PVML expressly, and only, communicates program state information the language could be said to have little to do with the actual visual representation that is generated. This definition is central to the architecture being proposed but also suggests that PVML is, in fact, a means to remotely debug programs in a variety of languages.

Accordingly, the proposal will also be located relative to the domain of heterogeneous distributed debugging. This leads to a definition of PVML as an imperative debugging language and to the PVML-based system being an abstract debugger.

Chapter 6 develops a set of requirements for PVML that is founded upon this definition of the language. Working from established approaches, a set of core requirements is developed. Some specialised extensions to the language are discussed, with a view to providing support for visualisation features that might support the particular pedagogical challenges posed by specific classes of language.

Chapter 7 describes the specific implementation of PVML that is presented in this thesis and justification offered for the decision to adopt an approach based on Extensible Markup Language (XML). The formal definition of this version of PVML is presented in the form of a Document Type Definition (DTD). Some examples are offered illustrating how PVML can be applied to a variety of scenarios.

Chapter 8 describes a significant part of the work undertaken – the creation of reference implementations for the target and engine between which PVML

flows. A brief description is offered of the reference engine and two reference targets against which PVML concepts have been evaluated.

Chapter 9 discusses and assesses the research undertaken and pays particular attention, given the open nature of the architecture supported by PVML, to future possible developments that could widen the application of the language.

Chapter 2

A LOCATION AND LANGUAGE INDEPENDENT NOVICE PROGRAMMING ENVIRONMENT

A novice programmer may well have a formidable task ahead of them. Learning to program a computer involves many new conceptual hurdles and a possibly difficult new set of mental models. However, in many cases, the environments through which computers are programmed have significant complexities in their own right. It would seem desirable to maintain a focus, for the novice, on programming language skills and considerations whilst minimising the distractions of mastering the environment that is being used.

Selecting or modifying a programming environment such that novice programmers are well supported is an acknowledged problem which this chapter sets out to examine. The assertion is made that an environment that is location-independent, language-independent and which offers program visualisation features would be a useful contribution to the field. This assertion will be critically examined in light of developments in the published literature. As will be seen, the requirement for a PVML arises in the specification of a programming environment that has these three characteristics.

This chapter maps a context for the work that follows, which focuses more precisely on issues relating to PVML. An exhaustive coverage of literature relating to the needs of novice programmers is not attempted here – rather signposting is provided that leads to the areas, relating to program visualisation, that are more substantially covered in later chapters. As stated in Section 1.2, an important motivation for this research is the provision of PV in a distributed, multi-language, novice programming environment.

2.1 Summary of motivation

The case is made for a certain style of programming environment for novice programmers. Drawing from literature relating to novice programmers, the assertion is made that such students would benefit from an environment that was portable (between home and school) and that supported the learning of multiple programming languages. It is also asserted that inclusion of program visualisation features should be considered. A discussion of the evidence that supports this view is presented in Section 3.7. It is the suggestion of providing PV features in a distributed, multi-language environment, that historically, and architecturally, gives rise to the PVML concept.

The specific focus, in terms of defining 'novice programmers', is a quest for a programming environment (PE) that adequately supports university first year – often referred to as CS1 or CS2 – programmers. To begin with, the term 'Software Development Environment' is explored in general terms, before seeking to define the distinct aspects that might characterise a learning environment as opposed to a production environment.

The needs of a novice programmer are shown to be distinct from those of a professional software developer. The survey presented here moves beyond the bounds of mainstream software development literature to address issues which are unique to the endeavour of teaching, and learning, programming.

The following issues motivate the directions taken in this chapter:

- Learning environments versus production environments

  Certain attributes of production programming environments render them unsuitable for the novice. At the same time consideration is given to features that would possibly only be used by a novice.

- Choice of programming language

  The controversial question of what makes a suitable CS1 programming language will not be addressed. Instead programming environments that support several different languages will be examined.

- Platform and location independence

  Programming environments can be complex to install and configure. The impact of this difficulty on student learning patterns is considered and a case made for an environment that transparently supports multiple locations – typically for a student, home and school.

As has been noted, this chapter stops short of discussing program visualisation, which is the main focus of this thesis. To a large extent the reasoning presented here has been published by the author in [106], to which the interested reader is referred.

2.2 The Software Development Environment

With a global economy, in which the production of software plays an increasingly important role, it is appealing to consider software development environments as the 'factories' of the 21st century. The software development environment, which will be referred to here as a programming environment (PE), provides many levels of support to the 'workers in code'.

Historically the means to support software development began to mature in the Unix operating system [54] as a set of utilities that communicated with each other via text files and the notion of 'pipes' – a primitive form of inter-process communication. The consequent interoperability of discreet tools gave rise to programming toolkit environments with support even extending to entire 'environments' based on the integrated use of such tools [31].

The increase of computing power available at the desktop, and the consequent development of the graphical user interface (GUI), created a shift in emphasis from the integration of low-level tools towards integration of the user interface behind which these tools operated. In this context the language based environment arose, in which the PE provided, within a single user interface, integrated access to all stages of the development process for a particular language. Lisp [93] and Smalltalk [33] were early beneficiaries of this approach.

2.2.1 Features

It is instructive to review the types of features that might be found in a modern professional PE. The list of features that follows is not intended to be exhaustive but the breadth of aspects that may be covered is an indication of the importance of the PE:

- Code writing support

  Most PE's provide an editor that is programming language aware at some level. The editor might, for example, highlight the reserved words of the language, match closing braces with opening braces or check basic language syntax.

- Code management tools

  In a large software development project many programmers are working concurrently on different parts of the source code. A mature PE needs to provide version control so that change is managed in a consistent manner. This support may be provided within the PE or as an external system with which the PE interacts.

- Tool Launching

  In the course of software development a variety of tools need to be used, such as a compiler, linker, profiler or debugger and the PE may provide an interface to these lower level tools enabling them to be invoked and configured from a single interface.

- Debugging support

  The actual process of debugging may be specifically supported by the PE which may provide data inspection and visualisation tools.

2.2.2 Software Process

The software development environment is a focus of study in its own right. Notkin [73] discusses the relationship of PE's to the various software engineering paradigms and process programming languages [2] are designed to abstract the process and formalise the design of PE's.

Support for a coherent development of software development tools has given rise to standards such as the Portable Common Tool Environment [12] which provides a standard for the 'backend' with which software development tools necessarily interact.

Analysis of the software development process at this kind of level gives rise to programming environments that have an ever increasing level of sophistication.

2.2.3 Learning Environment versus Production Environment

So far the discussion of PE's has focussed on the production-oriented needs of large software projects. In the context of this proposal attention must be turned to the needs of the novice programmer.

Jimenez-Peris has suggested [47] that an environment which supports the process of learning to program needs to include new features, and exclude existing features, relative to a production oriented environment.

The additional inclusions are directed towards the fact that the student programmer requires a greater level of abstraction of program structure and function in order to gain insight into their efforts. They may also need more assistance from the environment when they need to debug and correct errors.

Exclusion of features needs to occur in order to reduce the complexity and scope of the environment. For a novice the learning curve imposed by the development environment has the potential to eclipse the learning of a programming language. This can be related to the sheer size of the environment. Size can be quantified in terms of the complexity of the interface and the richness of the feature set provided by the environment. The Microsoft C++ development environment (not, by production standards, a large environment) offers the user over three hundred separate options and menus.

The author's professional experience particularly supports this line of reasoning. A complex PE (Visual Age for Smalltalk – an IBM production development environment) was used for several years to introduce novices to the Smalltalk programming language. It seemed clear that, in many cases, CS1 became a course in Visual Age rather than one in Smalltalk. The subsequent adoption of the Java language, along with an environment specifically designed for novices, BlueJ [57], has mitigated the situation. Even so, students who are struggling to learn Java, still find mastering the environment a barrier.

2.3 A Novice Programming Environment

Given the complexity of mainstream environments, and the distinctive needs of the novice programmer, it is reasonable for Jimenez-Peris to have suggested the removal, as well as the addition, of certain features. In these terms, the PVML proposal constitutes a significant addition and this chapter seeks to elaborate upon that context – a PE that provides features that explicitly seek to target the novice programmer.

The two aspects, language and location independence, are examined separately since they are important aspects of the architecture into which PVML fits.

2.3.1 General Features to Add

Although production PE's may on occasions implement some of these listed features they are not considered to be central to the formal software development process whereas the arguments for their inclusion in a novice programming environment are much stronger.

- Visualisation

  The question of program visualisation is most comprehensively addressed in Chapter 3 and so will not be discussed in depth here.

- Intelligent tutors

  The help system of a complex program can be as intimidating as the program itself. Novices will often not know what search terms to use within the help system, since they lack a mental model of what they are trying to accomplish.

  Work has been done on help systems that embody Artificial Intelligence (AI) such as the Lisp Tutor [1] and Pascal-based Proust [51], in an effort to develop a help system that understands what the novice is trying to accomplish.

- Informative Error Messages

  The novice is likely to spend more time looking at error messages than the professional programmer yet these messages are often expressed in terse, formal terms that are not helpful to novices. Error messages arise as a result of program syntax errors – these must be understood, and corrected, before an executable program is produced. The eventual execution may also generate error messages.

Explanations, and examples, will assist the novice but perhaps at the cost of execution efficiency in the PE. It has been noted [47] that the execution efficiency of a novice environment is of less relative importance.

- Language Aware Editors

  For a novice the language aware editor, described as a generic feature for a PE, is particularly helpful. Through the highlighting of language syntax and program structure such an editor offers support for one of the main hurdles for a novice programmer.

2.3.2 Professional Features to Remove

The simple answer as to which features of a professional PE would be appropriate to remove for the novice user, is 'many'. The sheer number of features alone, in a professional PE, act as a deterrent to the novice programmer. Although Eisenstadt and Domingue [21] have argued for a 'cradle to grave' PE, such an environment would need to implement multiple operating modes, which corresponded to differing levels of experience in the programmer.

The breadth of features that may be found in a professional PE, even though each may be strongly argued for as an inclusion in CS1 education, have the combined effect of deterring novices who have yet to write their first program in any language.

Features such as version management, multi module source management (`make`), group work management, software testing and specification tools should be removed, or at least made optional, in novice programming environments.

2.3.3 Language Independence

Much literature, that relates to the philosophy and pedagogy of computing science curriculum design, discusses programming language issues. There is considerable focus on the relative importance of teaching programming formalisms, compared with a more pragmatic approach driven by the current needs of the computing industry. What this leads to, at some level, is the choice of a first (and maybe second) programming language in computing degrees.

Whilst the choice of a first programming language can colour the overall theoretical approach in a computing degree, the choice of a PE is a logically separate and less extensively discussed issue. Curricula that are 'multi lingual' (as many are) usually require students to learn to use more than one PE.

Hendrix observes [41] that this has a tendency to lead students towards learning PE's rather than programming languages The Hendrix GRASP environment, "constructively" supports novices in a number of programming languages (currently C, C++, Ada, Java and VHDL - a hardware description language). Constructive support in this context is the ability to syntax check and pretty-print the student's source code. The key observation made by Hendrix is that it is "...the learning curve associated with environments, not languages that is the most frustrating to students".

An environment that supported all the languages that a student was required to learn, would be one with which the student would become very familiar. As latter languages were undertaken, the environment would become a support and encouragement for the language-learning process rather than a distraction.

2.3.4 Location Independence

The other novel aspect that has been considered central to a novice programming environment is the delivery mechanism.

Novice PE's that have been developed to run on Unix or Windows. The assumption is always that the student is seated at a workstation on which the programming environment has been installed and configured. This places physical constraints on where the learning may take place. Prior to using the PE at a particular location the PE software must first be installed at that location.

These constraints have the potential to vanish if the environment is delivered within a web browser. Literature relating to use of the World Wide Web as a vehicle for educational delivery has therefore been reviewed.

Boroni [10] describes the shift to web-delivery as a 'new paradigm' in education and notes the following features within that paradigm:

- That students are able to maintain a dynamic involvement with course material outside of the traditional dynamic experience – the face-to-face lecture.
- That lectures themselves suffer from not being repeatable – especially not being correctable if an error occurs – whereas web delivered material can be constantly refined and reviewed.

These points both relate to the delivery of standard course material through the Web. The presentation of more dynamic scenarios, material that was normally restricted to institutional computer laboratories, is covered in literature relating to Web-delivery of animations. In an earlier paper Boroni notes [11] that Web delivery enables animations to be used "without even the hassle of local installation required of most current systems".

An excellent overview of the area of animation delivery through the web is provided by Naps et al in [71]. Though this report primarily addresses visualisation delivery, the prevailing emphasis on visualisation of program execution, means that it is reasonable to generalise to the provision of PE's through the Web.

Naps suggests a taxonomy of Web visualisation delivery mechanisms, parts of which can be generalised to a Web-delivered PE.

The principle axis concerns whether the computation of a visualisation is remote or local (at a server or in the browser).

At one end of this continuum he identifies visualisations which are entirely downloaded to the browser. In this scenario the program that is being visualised executes at the browser along with the visual display. Such programs would necessarily be written in the Java programming language and it would be quite practicable to develop a PE for Java that functioned in this way. Jeliot [36] is an example of this approach, although not characterised as a PE.

This taxonomy describes an intermediate level of visualisation delivery, in which the execution component is downloaded through the Web into some locally installed, non-browser, packages such as a C compiler or spreadsheet. This model would require the host environment to be installed and configured at the user's computer. Naps notes the support problems involved and in the current context the aim of delivery being browser-based has already been suggested.

The 'remote' end of the axis is characterised as involving 'remote-run' and 'distributed-run' visualisations. This approach is the one that most closely corresponds to the architecture that the author has described in his Chiba paper [106]. The 'natural' division would be to run the 'model' on the server and the 'viewer' in the browser. In terms of a PE model translates to the program being written and executed whereas viewer represents the user interface of the PE.

2.3.5 Conclusion

The review of requirements for a novice programming environment has been brief. The substantial work in this thesis relates to program visualisation which is given a deeper treatment in Chapter 3. The intent to investigate PV systems

in which the target and engine are substantially decoupled, has arisen however, in the historical context of the architecture described – namely a programming environment that runs the target program on a server and the GUI in a web browser. The intent of this chapter has been to give that architecture some background within established work in the field.

The proposal for a PVML has arisen in the context of this suggestion for a distributed, and language independent, novice PE. The consequence of a convincing implementation of PVML would be to provide a basis for the implementation of the type of novice programming environment described.

Chapter 3

PROGRAM VISUALISATION

The starting point of this review is the assumption, as explored in Chapter 2, that a novice programmer can benefit from a programming environment that is explicitly designed for them. As has been suggested, such an environment may well be designed to be location and language independent. Such a tool could be conceived as being central to the early years of a computing science degree. The question addressed in this review is whether there is a case to be made for including program visualisation facilities in the tool.

The review starts by analysing the visualisation field in terms of several well-established taxonomies before moving on to assess the evidence for PV being beneficial for novice programmers. Particular attention will be paid to those aspects which relate to a model of PV that could incorporate a PVML-like concept.

PV has been briefly defined in the introduction as the technique of presenting visual representations of the execution of a computer program in order that its behaviour can be better understood. This understanding may be from the point of view of specific aspects of the program or more generally at the level of establishing mental models of program execution. At this stage it is necessary to look more deeply into this definition.

The term *visualisation* has many connotations in common parlance but the particular definition of this word that is at issue here is the one which suggests that a mental model of some concept is being formed. The psychological process of building a mental model[72] [92] of a complex process is an obvious step in understanding that process and the model-building can be aided by visualisations of that process.[94] These visualisations may in fact be 'visual', in the sense of a "picture being worth a

thousand words" or they could take other forms. A verbal representation such as a metaphor could be an aid in visualisation if it helps engender a new mental model. Sounds that are produced to correlate with some aspect of a complex behaviour might aid understanding – although this would strictly be termed 'auralisation' it is still, in the general sense being discussed here, an aid to visualisation. Other senses have yet to be explored – perhaps 'aromarisation' awaits the world!

The application of visualisation techniques in the field of computer software gave rise at a very early stage [82] to the term *Software Visualisation* and this association of the two words implies any technique that aids in the understanding of a piece of software. The term can refer to a process as straightforward as the organised presentation of program source code [56] or to one as sophisticated as the movie "Sorting out Sorting" [4] that portrays a selection of sorting algorithms using sound and vision.

There is an acknowledged division within the broad category of software visualisation into *Algorithm Visualisation* and *Program Visualisation* and the genesis of this division will be explored through the visualisation taxonomies that are discussed. The essence of this distinction rests on the level of abstraction of the raw program execution that is being offered – *Algorithms* are the higher level processes that are implemented by *Programs*.

As this discussion of software visualisation unfolds there will be a number of related fields of endeavour that need to be set aside and clearly defined as being beyond the scope of this research. The following terms, though at times referred to in the taxonomic literature reviewed, are being deliberately set aside:

- Visual Programming

  The reversing of the order of these two word stems describes a distinct endeavour. Typically programs are written by using a text editor to create and modify program source code. A visual programming environment enables the programmer to create and modify a program by manipulating graphical objects that represent fragments of source code syntax. Closely related areas, that will also be set aside, are 'Programming by Example' and 'Programming by Demonstration'. The focus in this thesis will be on visual techniques for understanding, rather than producing, programs.

- Computation Visualisation

  Visualisation techniques, which can be applied to the clarification of almost any process, potentially come under the umbrella of software visualisation when what they visualise is the process of computation. The use of visualisation to represent the performance and functioning of the underlying computer system (also termed 'Performance Visualisation') will be set aside. The focus here will be on the use of visual techniques to understand programs in a nexus that involves their creation rather than their eventual deployment in an actual computer system.

3.1 Taxonomies

This section will review the recognised taxonomies of software visualisation. Through examination of the work of Myers [68], Brown [15], Stasko [100], Price [83] and Roman [88] a focus will be developed on the particular category that is being addressed in this proposal.

There is a circular aspect to the presentation of a set of taxonomies – such a presentation, in reality, represents yet one more taxonomy. It is not the

author's intention to present another taxonomy but rather to justify an approach for the subsequent chapters of this thesis that is founded in a particular reading of the taxonomic literature available. In order to lead the reader towards this synthesis, this review has its own structure that relates to the conclusion being sought and rather than analysing each taxonomy in turn will present the major issues that are considered important and relate them to the literature.

There are a few concepts to which the reader may require an initial introduction to in order to facilitate understanding of the discussion that follows and these are presented as visualisation axioms in the section that follows.

The question of when to refer to the endeavour as software visualisation (SV) and when to use the term introduced earlier, program visualisation (PV) is one that is be discussed, at length, later in this review. To begin with, the more general of these two terms, SV, will be used.

3.2 Visualisation Axioms

Two aspects of the discussion of visualisation are considered so fundamental that they will be given a cursory examination before the full analysis is offered. These aspects are:

- The various human roles involved in the visualisation domain
- The distinction between static and dynamic visualisation

It is hoped that the brief coverage offered here will assist the reading of the more detailed analysis that follows.

3.2.1 Visualisation Roles

In the course of a discussion of software visualisation there will be cause to refer to a visualisation system from various, human, points of view. These points of view represent the roles of the various human agents that are

required to design, build and then use a software visualisation system. The explanation offered is that of Blaine Price [83], but there is little disagreement on this matter in any of the taxonomies presented.

The roles considered will be those of:

- Programmer

  The person who wrote the program that is being visualised (referred to here as the target). As Price observes the programmer might not have been aware that their program was to be visualised and they also may not ever witness the visualisation of their program.

- SV Software Developer

  Also, as Price notes, a programmer but in this case the program that they wrote is the one that enables other programs to be visualised.

- Visualiser

  The person who used the SV system to design and build the particular visualisation that is being considered. Often this role and that of the SV developer may overlap but the important distinction that is made by this separation is the relative involvement with cognitive rather than technical programming issues.

- User

  The person for whose benefit the visualisation is presented. The effectiveness of a SV system, or the particular visualisation being viewed, would be gauged by its effect on the user's understanding of what is being visualised.

It should be emphasised that these roles are not necessarily held by distinct, human players. In some cases a single person may wear more than one of

there 'hats'. Clarifying the roles is important because they embody distinct areas of concern in approaching any SV system.

3.2.2 Dynamic vs Static

The explanation of the distinction between *static* and *dynamic* visualisation offered here is based on the work of Brad Myers [68] but again these terms are so fundamental, and also largely un-contentious, that they are to be found throughout the literature. Dynamic visualisation refers to an approach that offers an evolving view of a program running – in effect a movie. A static visualisation offers still images that represent the program from time to time.

The emphasis in this review is on the visualisation of program execution and, as will be seen, the distinction between static and dynamic needs to be applied to most aspects of that discussion. The important point that is being made concerns the extent to which the display of the visualisation proceeds automatically (dynamically) or else is one that requires the viewer to select and view separate steps within the visualisation (static).

3.3 Program vs Algorithm Visualisation

Myers [68] presented "Taxonomies of Visual Programming and Program Visualisation", a categorisation that many consider underpins the field. He defines two axes along which to organise SV systems. Myer's first axis describes the extent of animation in the visualisation – static or dynamic as discussed above. The second axis, the one that is considered here, considers the extent of the abstraction of the program represented in the visualisation. Myers segments the axis according to whether it is *code*, *data* or *algorithm* that is being visualised.

3.3.1 Code Visualisation

Code visualisation refers to techniques which focus on the program source code. A static approach to code visualisation could be as straightforward as

the flowchart [37], perhaps the ancestor of all SV, or more recently attempts to increase readability of code by use of typographical techniques such as fonts and indenting [56], sometimes referred to as 'pretty-printing'. Each of these techniques seeks to expose the higher level structure of a program in order to assist the user in visualising that structure. The static approach does not offer a temporal axis – the user must provide this by tracing through the representation.

The dynamic approach relieves the user of this responsibility by stepping through the code as the program is executing and highlighting the code that is being executed. BALSA [14], often considered a seminal PV system, would pretty print the Pascal program source code in a window with the highlight moving as the program executed. Each call to a new procedure or function would cause a new source code window to open, providing a very direct visualisation of the call structure of the program. A programming language with a different execution model, such as Prolog, receives a different, but analogous, treatment in TPM [23] where a tree of Prolog predicates unfolds on the screen as the program executes.

3.3.2 Data Visualisation

The classic blackboard diagrams drawn by computer science lecturers teaching data structures – boxes with arrows joining them and values written within – are, in terms of Myer's taxonomy, static data visualisations. They are obviously static and they represent the storage of data within a program. His own Incense [67] system would automatically generate such pictures.

When the executing program itself is able to dynamically generate and update such displays, this has become a dynamic data visualisation. It is at this point that it would be reasonable to suggest that the execution of the program is being *animated*. There are numerous systems that provide dynamic visualisation of program data. One of the earliest, BALSA [14], already cited for its dynamic code display, implemented a second set of windows in which

representations of data structures were displayed as the Pascal program executed.

3.3.3 Algorithm Animation

The visualisation of data structures, described above, shows the content of actual program variables. For example a linked list may appear as a series of boxes joined by arrows. Although such a display may prove invaluable to a programmer that is having trouble writing code, it provides absolutely no information about the purpose for which the linked list is being used. A wide variety of computing science problems can be solved using the list as a tool, but the higher-level structure of the problem, for example whether the list represents a collection of bins or a tree structure, remains obscure. In order to display this higher level a system of algorithm animation is required.

ANIM [7] automatically generated such displays from programs written in a variety of source languages. The output was in a series of printed representations – hence this is an example of static algorithm visualisation.

The dynamic approach is again well represented. BALSA offers this level of display based on special instructions added to the program code. ALLADIN [43] allows the visual representation to be specified at run time by selecting and specifying graphical events. TANGO [103] adds gradual transformations to the visual sequences that are specified by adding special instructions to the code.

3.3.4 Discussion

The broad distinction between visualising programs and algorithms, as described by Myers, has been quite closely followed in the other taxonomies reviewed.

Marc Brown, in [15], defines 'Content' as one of three dimensions. He formalises the level of abstraction concept by considering whether the visual

displays map directly to data structures within the program. In a 'Direct' display the program data structure could be deduced from the display which is in contrast to a 'Synthetic' display, where the graphics portray more abstract concepts, which map to a higher-level, algorithmic view of the program. The distinction between program and algorithm remains, except it has been restated and refined as that between direct and synthetic.

Blaine Price, in [83] also uses the term 'Content' and makes some interesting observations about the line between 'Algorithm' and 'Program'. Whilst he regards algorithm visualisation (AV) as being "designed to educate the user about a general algorithm", he considers it "more likely" to be program visualisation, when a particular implementation is the focus of study. He adds that the provision of a view of program code in the system would lead to a program visualisation categorisation.

Price's taxonomy is distinguished by defining terms beneath the major top-level distinctions such as 'Content'. Indeed the whole Price taxonomy is designed to be extensible and is presented in the form of a tree of concept nodes. In the particular case of 'Content', Price directs further attention to 'Fidelity and Completeness' and 'Data Gathering Time'.

The definition of 'Fidelity and Completeness', in which Price cites Eisenstadt [24], seeks to explore the faithfulness of the mapping from program to visualisation. Price considers the extent to which a visualisation system displays the "full and complete behaviour" of the target program. He suggests that a "hand-crafted", algorithmic visualisation would have a low 'Fidelity' rating since few deductions could be made, from the visualisation, about the state of the underlying program.

'Data Gathering Time', as an aspect of 'Content', depends on whether run-time information, such as the values of data, is part of the visualisation. There is no connection between this aspect of 'Content' and the question of whether algorithm or program is being visualised.

To summarise, Price maintains the distinction between program and algorithm but adds to it in ways that shift his analysis beyond the structure being suggested herein.

In [88] Roman remains faithful to the Myers approach but splits along two axes within this area. 'Scope' is taken to define attention to a program's "code, data and control states, and its execution behavior", terms which neatly span the Myers distinction between code and data.

The second Roman axis is that of 'Abstraction', within which he squarely sides with Brown's 'Content' definition in citing the level of abstraction of the graphical forms, relative to the program code. The Roman taxonomy is useful here because it draws attention to the fact that Myers has set out to define an axis with two ends (program and algorithm) and has proceeded to mark three points on that axis (code, data, algorithm). It seems reasonable to split these issues in the way that Roman does.

This is borne out by John Stasko in [100], where he takes an approach that is roughly equivalent to Roman in identifying 'Aspect' and 'Abstractness' as two out of his four axes. Stasko uses the term 'Aspect' to define "a different aspect of a program... most clearly representing the purpose of the visualisation.... what parts of the program are being emphasised". The purpose of this term is to draw attention to what is being visualised rather than how it is being represented.

This latter issue is characterised by the 'Abstractness' axis which can be applied to code, data or algorithm visualisation and attempts to characterise the extent of abstraction. The example Stasko uses to clarify this point is a representation of time in a program. A non-abstract ('Direct' in Brown's vocabulary) representation would show the variables in memory and their values –- hour, minute and second or possibly just a large, binary number. An abstract display ('Synthetic' to Brown) might display a picture of a clock face. As a key to deciding whether it is 'Abstractness' or 'Aspect' that are

being determined, Stasko offers the term 'intention content' to refer to the extent to which a visualisation attempts to expose the meaning behind code or data.

A visualisation with a low level of intention content remains close to the raw data structures in the target program. A greater level of intention content in a visualisation, displaying more abstract views, entails active effort on the part of the visualisation system and its designer. In terms of the clock example, it is the introduction of the intention content "telling the time", that leads to the effort to present a clock-like display rather than a low intention content representation of three integers.

3.3.5 Conclusion

The desire to categorise the extent of abstraction is one of the fundamental issues in all taxonomies of visualisation and the broad terms 'program' and 'algorithm' are ones that have wide acceptance. The reason that this issue is of such importance is that it profoundly influences the extent to which visualisations can be automatically generated by straightforward means.

The more a display gravitates towards the algorithm end of this spectrum the more likely it is that there will need to be human intervention in deciding what the intention content really is and how that can be mapped to a visual display. Producing a visual representation of a program execution implies definition of a distinct set of mappings from the states of the program to some form of visualisation. The generation of this mapping is the issue that is examined next in an examination of the topic of automation in visualisation.

3.4 Automation in Visualisation

The topic of automation in visualisation focuses attention on the process by which the visualisation is generated – categorising the extent to which the visualisation simply 'happens' as a side effect of program execution in

contrast to a display that requires effort on the part of one of the participants.

The Myers taxonomy, though still the starting point for many discussions in the field of SV, has little to say on the topic of automation. His two axes – dynamic-static and program-algorithm – are the full extent of his categorisation. At first thought it might seem tempting to associate degrees of automation with varying positions on the static-dynamic axis, but this axis is intended to categorise the presentation to the viewer rather than the way in which a visualisation is derived. It is quite conceivable that a static visualisation be automatically generated (Incense [67]) or that a dynamic visualisation be generated by hand (Sorting Out Sorting [4])

To clearly locate the issue of automation in visualisation, attention must be turned to the other available taxonomies, all of which make some reference to this aspect.

The hierarchical approach taken by Price in his taxonomy [83], defines 'Method' as a top-level category and beneath this divides between 'Visualization Specification Style' and 'Connection Technique'. Each of these intersects with automation to a certain extent.

The question of 'Visualisation Specification Style' essentially asks how the content of the display is derived. On the one hand the display, even if dynamic, may contain a completely fixed set of events that were determined by the SV designer (Sorting Out Sorting) – in this case the specification style is *fixed*. At the opposite end of the spectrum, debugging type environments such as TPM [23] and Lens [65] automatically generate displays with no explicit intervention. Between these two extremes lie many systems where the programmer or the visualiser (roles as defined earlier) can specify the form of the display. For example in TANGO [103], the programmer may add statements to the source code to cause interesting events to have visual consequences. Indeed Price draws attention to the fact that "automatic

systems have the advantage of making the programmer, visualiser and user into the same person" (at least potentially)

The term 'Connection Technique' refers to a slightly separate issue – the manner in which the assertions concerning the visual display are relayed "between the visualisation and the actual software being visualised". Some aspects of this are barely distinguishable from the question of visualisation specification style. For example, when annotations are added to a program in order to control a display, a style of specification is being employed that lies mid way between fixed and automatic. At the same time this is the technique of connection through which the visualisation is driven by the program. The terms defined by Price become confused.

Other concerns regarding connection technique are more clearly distinct – for example:

- Does the target need to run at the same time as the visualisation is viewed?
- Do target and visualisation need to run on the same computer?

These questions are very pertinent to the core of the PVML proposal, which in Price's terms, could be characterised as a PV/Method/Connection Technique proposal.

Brown's [15] taxonomy begins with a definition of three axes for the categorisation of displays:

- Direct/Synthetic

  As already discussed this maps to program/algorithm

- Current/History

  Describing the timeliness of the display – namely whether it shows past states as well as the present state of the program.

- Incremental/Discrete

  Defining to what extent changes in the display simply happen (Discrete) or are represented as a transition (Incremental)

None of these particularly relate to the issue of automation but the bulk of his paper does in fact discuss automation, applying the taxonomy described above to systems that are capable of automatically generating displays. He draws particular attention to the fact that steps in an algorithm execution may not usefully map to discrete access to the program data structures. This issue, that of automatic algorithm identification, is discussed in detail below.

Stasko [100] also made automation one of his four top-level categories ('Aspect', 'Abstractness', 'Animation' and 'Automation'). Bearing in mind the earlier discussion of Stasko's taxonomy, in the program versus algorithm visualisation section, a key observation is that "our abstraction and automation dimensions usually exist in an inverse relationship. Creating program visualization views with high levels of abstractness involves a great deal of intention content and simply requires a priori design support". This leaves the field of automatically generated visualisations populated on the whole by straightforward, low-abstraction program visualisation. Notable exceptions, such as UWPI [42], have a restricted domain of operation – only generating certain pre-defined types of visualisation.

In [88] Roman associates the automation of visualisation generation with his category of 'Specification Method' which "encompasses the means whereby the animator specifies which aspect of a program are to be extracted and how they are to be displayed". He decomposes the specification method into a series of broad types of technique and these will form the basis for further discussion as they relate closely to the PVML proposal.

- Predefinition

  This is a fixed mapping between program state or events and the display as implemented by a variety of debugger style environments. The user is not given the opportunity to modify the semantics of the display and there is no input from the visualiser role.

- Annotation

  Annotation of the programs being visualised is the predominant technique for imposing the visualiser's will on the display in the cases where a higher level of abstraction from program code and data is required. The technique of annotation was pioneered in Balsa [14] but has been used extensively since. Sometime referred to as the technique of *interesting events,* the annotation, inserted into the program by the visualiser designing a visualisation or by the programmer seeking to expose program behaviour, has the effect of updating the display in some manner. There are many consequences of this approach and it is discussed in more detail below.

- Declaration

  Although an organised approach to annotation would result in certain states having certain visual analogues the overall mapping is not clearly, and independently, defined. The declarative approach takes as its starting point a definition of a set of program state/display mappings and then arranges that the visualisation system is simply kept aware of program state. Roman has used this approach in Pavane. [87] This is a dramatic departure from the architectures described so far because it clearly decouples the program state from the visual consequences. The work of Roman will be examined later in this review when the focus is on this decoupling and the implications for the PVML proposal.

In terms of the approach to PV that is being suggested in this thesis there are two aspects relating to automation that need to be taken further:

- The nature of annotation and its consequences both technically and pedagogically for the novice programmer
- The current state of automated algorithm identification.

3.5 The Annotation Issue

Annotation of the target program source code is one of the predominant approaches to creating visualisations of that code. Price [83] specifically reserves this term to refer to a system where the additions to the program source code are hidden from the programmer[1] by a special editor.

At the level of the version of the code that is executed and visualised, annotation involves modifying the program to include procedure calls that give rise to visual behaviour. Embedding these hooks in the flow of the program is described by Robert Henry [42] as *control intrusive*. If the programming language provides a means to attach such hooks to data structures they might be termed *data intrusive*.

The selection of where to make these calls involves decisions about which steps within the program execution give rise to interesting events. For example the incrementing of a loop counter may only be of interest to the overall aim of the program when a certain, critical, comparison is made. The success (or failure) of the comparison is an interesting event, whereas the incrementing of the loop counter is not. As can be seen the choice of what is interesting requires a higher level comprehension of the program algorithm. What happens visually when an interesting event occurs is a question that involves issues of the visual psychology of the user of the system.

[1] Price uses the closely related term *instrumentation,* to describe explicit addition to the source code

The default state of an unmodified executing program is that no events within the program are signalled externally, other than at points when the generation of output is explicitly part of the program. If this same program is recompiled with the option to debug the program execution, the program is able to be executed step by step, a line of source code at a time. Running a program under a debugger is rather like instrumenting every single line of source code since the debugging environment can readily be adapted to communicate each, or selected, steps to a PV engine. This is the approach adopted by TPM [23] and DBX [6] and many other systems. Although no actual instrumentation has taken place, these systems could be characterised as potentially automatically instrumenting every single line of the program.

A second approach, that achieves the same goal, is to use a special compiler that instruments the generated execution module [95] without modifying the source code.

Instrumentation, whether automatic or not, suffers from one telling criticism – namely that instrumentation has the potential to change the behaviour of the program. In general, a program that employs a single thread of execution is unlikely to have its behaviour modified by instrumentation. However multi-threaded or concurrent programs, where the relative timing of events in several threads of execution can significantly alter program outcomes, cannot necessarily be safely instrumented. The debugging and visualisation of concurrent programs is a distinct area that requires further attention – but one which will be set aside within this proposal. It is intended that the techniques proposed here be applied to single threaded programs only in the first instance.

Manual instrumentation, which requires the programmer to, in effect, add procedure calls to interesting events, has been the subject of criticism on the grounds that this activity is extra work for the programmer [89]. In the case of a novice, the additional cognitive load imposed by instrumentation can

detract attention from the programming issues which ought, pedagogically speaking, to be the prime focus. In the case of an experienced programmer, the extra steps simply may never be taken since the focus is firmly on writing the program. These two points could be taken as arguing for an automatic instrumentation approach and it is worth considering a way in which automatic instrumentation can occur, in object oriented languages, without the need to use special compilers or debuggers. This technique, described below, exploits a fundamental property of those languages.

3.5.1 Class behaviour

Several writers [15], [68], in discussing annotation or instrumentation, have made the observation that an object oriented language is potentially self-annotating. The reason for this is that object behaviour can be *overloaded*. Although an object representing an integer is intended to participate in expressions involving other numbers, a specially modified integer object, that also understood how to visualise itself, could in fact be substituted. This special class of object, that would reproduce all the normal behaviour of an integer, also understands how to portray itself in a visual display. Such an object could be transparently used by the program in its default operation whilst, at the same time, being visualised. For normal execution the program would be given access to the unmodified integer class.

This is convincingly demonstrated by Jeliot [36], which is web-delivered visualisation system written for novice Java programmers. The novice writes Java code in a Jeliot applet window and submits the source code to the Jeliot server for compilation. The server compiles the code but employs instrumented versions of Java base classes. The resultant byte code (the executable form of Java) is returned to the novice's browser and, when executed, is able to visualise itself. In terms of the algorithmic goals of the novice programmer the target program is unmodified yet the special executable form is visualisation enabled.

This same approach is used in several other PV systems [22] [19] and has been usefully described by Thomas Naps [71] as making visualisation a "pure natural side-effect" of normal program execution. It is of interest in the context of the PVML proposal because it represents a very natural location for PVML generation to take place. The requirement for self-instrumenting classes in an arbitrary object oriented language would simply be that they described themselves in PVML.

3.5.2 Automatic Algorithm Identification

A brief overview is offered of literature that relates to the automatic identification of algorithms. Essentially, this is a topic that lies beyond the bounds of the PVML proposal, since deductions about what will be displayed in response to particular program states, in other words how *abstract* the display will be, are made by the occupant of the visualiser role. These take place after the PVML stream has delivered the program state to the visualisation module.

Nevertheless it is worth paying some attention to what Price [83] categorises as the 'Intelligence' of the 'Visualisation Specification Style'. Perhaps un-surprisingly, given the generally disappointing penetration of artificial intelligence (AI) techniques into the world of real systems, Price notes that "intelligence is sorely lacking among automatic SV systems". Automatic software visualisation systems are those that do not rely on human selection of interesting events .

The contributions made by AI to the automatic identification of algorithms is characterised as either *deep* or *shallow* depending on the extent of the constraints that are applied to the domain before the AI component takes effect. A completely open approach, that sets out to deduce the algorithm in an arbitrary program would be deep AI. One that operates within a set of constraints that limits the possible scope would be shallow.

UWPI (University of Washington Illustrating Compiler) [42] is a good example of a PV system that employs shallow AI and an examination of this system can yield some insights into the issues involved in automatic algorithm identification. The target program in UWPI is analysed by an *Inferencer* that looks at how variables in the program are being used. Abstract Data Types (ADT's) are inferred, based on a preloaded rule base of programming idioms and common ADT's. It is the scope of this rule base that limits the scope of algorithms that UWPI can recognise. The 1990 description of UWPI shows it being used to recognise a selection of sorting and searching algorithms.

A deeper, in AI terms, approach is represented by the Programmers Assistant [86] project from MIT. The Assistant, which is described as a project which "overlaps both artificial intelligence and software engineering", uses a formal representation of programs and their languages known as the *Plan Calculus*. Plan is described as a combination of "the representation properties of flowcharts, data flow schemas, and abstract data types". The approach is similar to that of UPWI but the library of *clichés* that is provided is more general, enabling a broader cross-section of algorithms to be identified.

3.6 Decoupling Visualisation

The earlier discussion of automation in visualisation paid special attention to the work of Roman and his identification of a declarative model of visualisation. In the context of the PVML proposal, this model is of particular interest because it clearly delineates the area of concern for the various visualisation roles. This in turn underpins the decoupled architecture in which PVML plays a part.

A more extensive examination of the work of Roman and others gives rise to a clearer enunciation of these issues and is the topic of Chapter 4 of this thesis.

3.7 Evaluating Visualisation

As has been stated previously in Price's taxonomy [83], the literature in which visualisation is systematically and objectively evaluated is far outweighed by that describing the development of yet another PV system. In the words of Price: "The most disturbing observation is the lack of proper empirical evaluation of SV systems, for if the systems are not evaluated, what is the point of building them?"

Most PV systems are developed by researchers with involvement in computer science education and the systems are used by students in those institutions. Generally, the evaluation of the effectiveness of the PV systems is anecdotal and experiential. There are substantial issues, that are germane to all educational research, that should really be taken into account in designing experiments to evaluate PV and these issues are often not the area of expertise of PV developers. The PVML proposal, as was stated in the introduction, is a suggestion as to how the PV evaluation domain may be opened up to researchers who have more expertise in educational research rather than in software development.

The PVML proposal is aimed at reorganising the fundamental architecture of PV and in doing so to enable a variety of existing systems to interoperate. This proposal does not make suggestions about how programs should look when they are visualised, nor about what aspects of the program could be most usefully visualised for novice programmers. If the PVML proposal were at such a level, there would be a strenuous requirement to review the literature relating to the efficacy of PV itself and through this to reassert the case for PV. Since the intent of the PVML proposal is to render PV more open to evaluation, this section of the PV review merely sketches the landscape of the literature that assesses and evaluates PV. No concerted case is made for the usefulness of PV.

Certain systems have undergone more methodical evaluation than others. Price singles out the work of Stasko [101] on TANGO and Goldenson [34] on Pascal Genie (which is a commercial development based on Incense [67]). These quantitative evaluations detected some benefit from PV, but in the TANGO study the benefit measured was not statistically significant. Other studies reported by Price are described as informal.

Mulholland [66] has characterised most evaluations of PV as "coarse grained". Such evaluations seek to measure the broad benefit of PV (relative to a lack of PV) or else compare two PV systems. He pleads for, and performs, 'fine grained' evaluation in which a detailed examination is made of the interactions between the students and the PV system. He subjects these interactions to 'protocol analysis' and by close examination was able to motivate quite specific improvements in the PV system that he developed.

It is the power of the cognitive research methods applied that leads to these very tightly targeted results and, in terms of the PVML proposal, the aim is to enable greater emphasis to be placed on the cognitive research methodology and less on the provision of PV systems.

3.8 Conclusion

The objective of this review, in the broader context of this thesis, is to set the scene for the ensuing, more detailed, examination of the issue of the decoupling of visualisation targets and engines. The reader has been drawn towards issues that underpin the concept of a decoupled program visualisation architecture, such as the roles of automation and annotation in program visualisation and the underlying issue of whether program or algorithm are being visualised. In the course of describing these preoccupations in the program visualisation field, definitions have been presented of many of the fundamental terms in the field and particular attention has been paid to the authoritative taxonomies of program visualisation.

In weighing up the taxonomies, paying special attention to the issues that are germane to the PVML proposal, greater weight has been attached to the taxonomic approach of Roman and subsequently, the work of Roman plays a central role.

The question of how the efficacy of PV for novice programmers has been evaluated to date has also been briefly examined. It is the lack of extensive empirical results that sustain or deny the proposition that PV is helpful to novice programmers that provides one of the fundamental motivations for the work described in this thesis.

Chapter 4

DECOUPLING VISUALISATION TARGETS AND ENGINES

This chapter begins with the assumption that PV facilities are to be added to the proposed novice programming environment. The focus in this review is on the ways in which that goal can be accomplished. In particular this review assesses the case for substantially decoupling the visualisation target, the programmer's currently executing program, from the visualisation engine, the components that provide the programmer with a visual representation of their program execution. The case is based upon the work of a number of researchers, who have identified approaches to visualisation that incorporate a decoupled methodology.

This review seeks to make a case for establishing a generalised communication protocol at the target/engine boundary – namely a Program Visualisation Meta Language. A central justification for the direction of this research is presented against the background of the earlier reviews. In conclusion, a very broad definition is offered, of how a Program Visualisation Meta Language fits into the generalised architecture of visualisation adopted by the literature presented here.

4.1 Where to make the cut?

At a general level, a case is being made for decoupling the executing program from the visual display, but as the figure below shows there are two steps along the way and the separation could be made at either of these points.

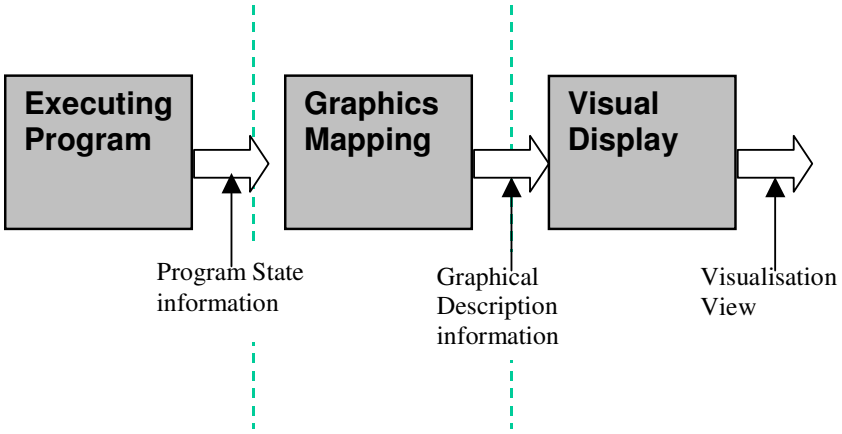

Figure 4-1 Potential visualisation decoupling boundaries

At the target end of the system there will be some means or other for the executing program to communicate its execution state. As will be recalled from the taxonomies examined, this might involve the annotation of interesting events or the program may be running in an environment which is able to automatically generate state information. By whatever means, the output from the executing program is in terms of its execution state.

At the opposite end of the chain, where the user sits, there is a visual display on which the PV is being viewed. The input to this display device is in the form of instructions that relate to graphical primitives. "Move the second box down", "Draw an arrow between the 6th & 7th boxes" are the kind of directions that the graphical display would be configured to interpret.

A split at this stage would require some form of *Graphical Language* to describe what is to be displayed.

The box in the middle represents the point at which a particular execution state or event is mapped to a visual representation. This is where Roman [90] has applied his declarative approach – as a middleman between the visualisation target and the display. Similarly it is where Domingue [21]

applies *views* and *mappings*. A closer examination of the work of Roman and of Domingue leads to a clarification of whether the boundary be one across which program state or graphical description information is communicated.

4.2 The Case for Decoupling

The survey of the PV field, presented in Chapter 3, has established the spectrum of terms that are used to refer to the various aspects of visualisation. In considering the question of decoupling the target from the engine, the focus needs to be on what Price describes as 'Method' in his taxonomy. He uses this term to refer to the means that are used to generate the display. Stasko refers to this as the 'Automation' axis.

The selection of a PV 'Method' is fundamental to the design of all PV systems. In human terms this is most clearly reflected in the precise roles of the PV players – the programmer, the PV developer, the visualiser and the user. PV that relies on manual annotation of the target program tends to combine the roles of visualiser, programmer and user since decisions about the nature of the display are being enacted by active modification of the target program and viewing of the results. If the PV developer has introduced a degree of automation into the control of the display, the visualiser role becomes trivial, since most decisions about the nature of the display are already made. The user or programmer will proceed to use the system with only marginal actions, such as selection from menus of representation styles and content, that could be seen as acts of a visualiser.

In terms of the argument being presented here, it is important note this observation – namely that the area of the system in which each role is active differs according to the method of visualisation that has been implemented. For example, a method that depends upon code modification tends to coalesce the roles of programmer and visualiser. The table below sets out to clarify this, identifying the location of the various roles in the context of Roman's three specification styles. Comparable tables could be drawn using

the other associated taxonomies, but the approach of Roman is the most straightforward in this regard. The table shows the way in which a particular human 'player' ends up enacting a variety of roles, depending on the precise specification method that is being used.

The table shows, for example, how the annotation approach has the effect of overloading the human programmer with visualiser activities.

Roman Taxonomy Categorisation		**Visualisation Role Players (Human Actors)**			
		User	**Programmer**	**Visualiser**	**Developer**
Specification Method	**Predefinition**	User	Programmer	×	Developer Visualiser
	Annotation	User	Programmer Visualiser	×	Developer
	Declaration	User	Programmer	Visualiser	Developer

Table 4-1: Who plays what role? Visualisation players and their roles in Roman's three specification styles

There are no clear, persistent, boundaries that define the areas of concern for the various actors, yet logically, in the terms that were used to define the roles, their concerns should be distinct. Reasserting the intention behind the roles that have been defined should make this clearer

4.3 Roles Revisited

The four roles, User, Visualiser, Programmer and PV Developer that were initially mentioned in the PV review are discussed in more detail here.

4.3.1 Programmer Role

The programmer has the goal of writing and debugging the program that is the target of the visualisation system. The concerns of the programmer are twofold:

- The overall, high-level description of what the target program sets out to achieve. This is the algorithmic description of the target program and is the fundamental starting point for all programming projects, although novices may neglect this area.
- The lower-level concerns of the particular programming language that is being used. How can the language features be used to implement the algorithm?

Some visualisation scenarios may not require a programmer at all. If the intent is to demonstrate algorithms, implemented by ready-written code, the introduction of programming language specifics will be a distraction.

The clear intent of the programmer role, when it exists, is to manage the program source content of the target program.

4.3.2 User Role

The user is the ultimate viewer of the visualisation. The entire purpose of the visualisation system is to assist the user in visualising a program or algorithm. It is the mental models of the user that are intended to be enhanced by the devising of new and better visualisation systems.

In the case of automatic, dynamic visualisations the user has little to do other than look at the display – perhaps controlling what is being displayed as a TV watcher might control a VCR. Static displays require the user to "turn the page".

As soon as the person looking at the display begins to make substantial decisions about the form of what is displayed they are beginning to enact the visualiser role in addition to that of user.

The aim of defining the user role is to isolate the consumption of the visualisation, as opposed to any part in its production.

4.3.3 Visualiser Role

In proposing the declarative method of program visualisation, the term used by Roman in his taxonomy, Roman defines visualisation as "a mapping from programs to graphical representations". This concept is of considerable importance to the PVML proposal and will be looked at in more detail later but at this stage it is also extremely useful in clarifying the role of the visualiser.

Given an executing program, and the goal of enhancing a user's mental model of the program, it is the job of the visualiser to design and modify the visual representations that will be observed by the user. The logical scope of the visualiser's activities is the nature of the mapping between program state and visualisation – namely the precise area encompassed by the Roman definition.

It is not the intention that the visualiser modify the program although some PV implementations may require this. Neither is it the intention, necessarily, that the visualiser interact with the display. A particular PV system might enable the user to participate in the planning of the display – in which case an individual actor will play the role of user as well as visualiser.

In becoming clearer about the nature of the visualiser role it is also becomes clear the way in which PV systems might become decoupled. The goal is that the tools and artifacts that the visualiser needs to interact with are distinct from other components in the system.

Since the visualiser is most in control of what the user sees, it is the visualiser role that intersects most with that of an educational researcher who is seeking to assess the efficacy of various visualisation approaches. If the activities involved in the visualiser role are adequately decoupled from the rest of the PV system then PV systems can be exposed to greater, and more methodical, introspection concerning their usefulness.

4.3.4 PV Developer Role

The activity cycle of the PV developer is one that should be the mirror image of the other three roles. When the PV Developer is active, adapting or correcting the PV system the user, visualiser and programmer will be idle. The PV developer role is the one that is least likely to overlap, within a single human actor, with the other roles.

Having clarified these roles the concept of declarative visualisation and its use in the Pavane [87], Vis [21] and ALADDIN [40] systems is expanded.

4.3.5 Discussion

As the definitions of these roles are being reasserted it is prudent to restate that a given human participant can, in a particular scenario, enact one or more of these roles. Table 4-2 demonstrates that precisely "who does what and when" will depend, not only on the visualisation specification method employed, but also on the type of scenario. Particular attention will be paid to the disposition of visualisation roles where novice and expert programmers are involved.

- Experts

  An expert programmer will be making use of the PV system in order to design and debug a complex program that they are developing. In this context the roles of programmer and user are likely to be predominant since visualiser activity, the design of representations, represent a significant distraction from the job in hand. The tendency of expert programmers to ignore PV systems, due to extra effort of enacting the visualiser role has been noted by several authors [65] [83]

- Novices

  A novice programmer may use the PV system simply as a user to observe the workings of algorithms. Where the novice is seeking to learn programming they will also enact the programmer role and in [104] Stasko draws attention to the positive motivational effect of asking novice programmers to be visualisers as well. He suggests that the program comprehension of student programmers was heightened by giving them the additional task of designing visualisations for their programs.

Visualisation Scenario		**Visualisation Role Players (Human Actors)**			**Comment**
		User	**Programmer**	**Visualiser**	
Experience Level	**Expert programmer**	×	Programmer User	*De-emphasised*	The expert programmer is focussed on the program
	Novice (studying algorithms)	User	Programmer *(prepared what is being watched)* Visualiser?	Visualiser?	The novice enacts the user role only. Who enacts the other roles depends on the system.
	Novice (studying programming)	×	Programmer User Visualiser?	Visualiser?	The novice is programmer and user. Stasko recommends being the visualiser as well! Who enacts the other roles depends on the system

Table 4-2 Who does what? Visualisation roles for different experience levels

4.4 Declarative Visualisation

The PV review presented earlier focussed on a number of accepted taxonomies of the field. These taxonomies have mapped out a variety of axes along which to categorise existing PV systems and the authors of the taxonomies have used these axes to locate their own work in the field. As has

been observed earlier the result is a plethora of distinct PV systems and a lack of any unifying concept that can lead to integration of these individual pieces of work.

Two authors have drawn attention to this gap, Roman in his own taxonomy of visualisation [88] and Domingue in his description of Vis, a novel visualisation system in [21].

Both of these authors have introduced a level of abstraction into the discussion of visualisation, by making the same broad assertion about the nature of the visualisation task. For Roman visualisation is "a mapping from programs to graphical representations" and for Domingue "events and states [of a program] are mapped into a visual representation". This concept of what visualisation is leads directly to a *declarative* model of visualisation which clearly defines the role and concerns of the visualiser as being the creation and manipulation of such mappings. When such a distinction is firmly enforced by the system it becomes clear that the visualiser has no involvement in the internals of the visualisation target execution. All visualiser activity is predicated on transforming some representation of program state into the new graphical form and it is precisely this that calls for, and supports, a clear decoupling of target and engine.

For Roman the formal definition of the declarative approach leads to *Pavane* which is described in detail below. Pavane establishes a language for declaring the associations between program state and pictures.

From the point of view of PVML the consequences of Domingue's work are even more interesting since he uses his framework of visualisation to create a meta PV system called *Vis*. Vis is actually a *SV system-building system*. By clearly isolating the visualisation mapping component, Vis is able to "reverse engineer existing PV systems and construct new systems with ease". This is very close to the goal of PVML.

4.5 The Roman contribution to visualisation

The review of PV literature that has been undertaken has clearly delineated the work of Roman as having particular relevance to the PVML proposal. Roman's published work in the field consists of an actual PV system, Pavane, and subsequently, significant introspection regarding the nature of PV.

4.5.1 Roman's taxonomy

Out of all the taxonomies described in the PV Review there is only one [88] that moves beyond the categorisation and description activities that are most usually associated with a taxonomy. The Roman/Cox taxonomy sets out to make a broad and formal definition of what visualisation is before beginning a categorisation that is viewed from the point of view of this formalism.

The definition of software visualisation suggested by Roman is that of "a mapping from programs to graphical representations", a suggestion that is clearly related to his earlier work on Pavane [90] which is described in more detail below.

By conceptually decoupling the visualisation from the program, Roman is able to create a division of labour amongst the four visualisation roles that have been defined. The Pavane system actually incorporates a tool that specifically targets the needs of the visualiser role without overlapping into the domains of any of the other roles.

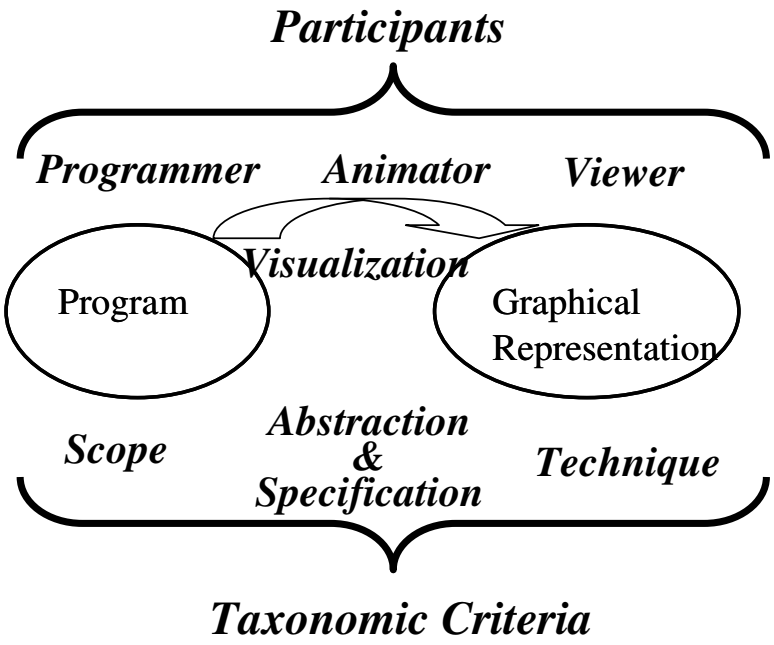

Figure 4-2: The Roman View of Visualisation. Reproduced from [89]

This diagrammatic view of visualisation presented by Roman makes these interrelations clear. In the diagram Roman shows:

- Participants

  The three participants that are involved in use of a PV system (as opposed to development of one). These have been referred to, in this thesis, as roles and have also used the term visualiser in place of animator.

- Activities

  The respective domains of activity for each of these roles – namely the program, the graphical representation and the transformation between them.

- Taxonomic Criteria

  The criteria from his taxonomy that are relevant to each role. 'Scope' is the term he uses to refer to the aspects of the target program that are to be visualised – in his earlier work on Pavane the term *domain* was used at this point. Similarly, when describing the graphical representations the term 'Technique' is used which had formerly been referred to as *range*.

This taxonomy and formal conception of what visualisation entails arose, historically, in the context of the development of the visualisation system Pavane, which is described in some detail in the next section.

4.5.2 "Pavane" - A Declarative Approach to Program Visualisation

Roman describes [87] an approach to specifying the contents of a visualisation which fundamentally decouples the target from the engine. In this proposal he suggests *declarations* that associate specific visual events with specific changes in program state. Hence, given a means for the target program state to be communicated, the visual consequences of that state are independently controlled by the set of declarations that have been established.

The essential aim in Pavane, the PV system he developed, is the 'separation of concerns'. The programmer is concerned with the writing and testing of program code whereas the visualisation of the execution of that code can be placed in the hands of a 'program animator' who does not necessarily need access to the program code. The Pavane system was used to visualise programs written in a concurrent programming language – Swarm.

In describing Pavane, Roman introduces a pair of terms, *domain* and *range* that are equivalent to the terms *scope* and *technique* that he went on to use in his

later taxonomic paper. *Domain* (*scope)* refers to "which aspects of computation are examined" whereas *range* (*technique)* refers to "what graphical objects and techniques are provided". Pavane provides the means to define mappings between domain and range.

He suggests that most existing PV Systems neglect to "explicitly implement a mapping" since they use annotation of the program code to single out interesting events and to request a specific graphical presentation of that event. The visualiser in such a system must identify points within the source code at which 'interesting' transitions take place. The visualiser would proceed to instrument the code with appropriate graphical calls.

Aside from the possibly confusing overlap of roles involved, there is a quite fundamental problem inherent in this *imperative visualisation* approach. The most useful display from the point of view of understanding the algorithm (a synthetic display in Brown's terminology) may need to represent a complex set of conditions within the program with a single visual metaphor. This is particularly so for the concurrent programs that Pavane seeks to visualise for an interesting event may be a "nebulous entity defined by state changes in a large number of discreet processes". However, this kind of statement can be made about any program in any language, in the sense that high-level, abstract concepts may have a complex relationship to the particular program language entities that represent them. The lack of a general, one to one relationship, between program execution events and the more complex 'interesting events' that are to be visualised, is the central justification for adopting a declarative approach to visualisation in Pavane.

The preoccupation of many of the taxonomies with the program/algorithm axis (direct/synthetic in Brown) can be restated to be a question of which of the many possible mappings between domain and range are to be defined. A single system can be characterised as either direct or synthetic depending on what mappings have been created, supposing that appropriate tools are made

available to the visualiser. These tools are "assumed to have complete access to the [program] state" in order that the visualiser has the freedom to declare any mapping they wish.

A key observation, from the point of view of locating PVML, is that the annotation of the running program which is necessary to provide this access "could in principle be largely automated" since "the entire state is examined rather than animator-defined events". In terms of the Roman model, PVML could be defined as an open protocol of automated annotation communication that can provide state input into a visualisation mapping process.

The bulk of the paper describing Pavane is concerned with the formal syntax used to define the relationships between program state and visual output. This language defines state, in either the *state space* or *animation space* of the visualisation, using collections of tuples. The detail of the mapping syntax and implementation is beyond the scope of this review. For the purpose of locating the PVML proposal, it is sufficient to note that PVML plays a part in the communication between what Roman refers to as the domain and the range of the visualisation.

4.6 The Domingue contribution to visualisation

The other major visualisation work that embodies a rigourous separation of visualisation roles is that of Domingue, who is a colleague of Price and Eisenstadt who have previously been cited. Domingue has not published explicitly taxonomic work that can be set beside that of Roman but the *Vis* system, which is described in some detail below, incorporates a similar architecture.

"Vis" - a Framework for Describing and Implementing Visualisation Systems

Vis [21] appeared the same year (1992) as the Pavane paper and hence Vis and Pavane would appear to have had little influence on each other's design.

Nevertheless they adopt a very similar approach to visualisation in that they both isolate the actual visualisation decisions to s specific layer in their systems.

The architecture of Vis considers "program execution to be a series of history events happening to (or perpetrated by) players." Domingue compares these *history events* to the interesting events that Brown spoke of and they represent some combination of program execution and data state.

The mapping of history events into a visualisation is handled by two subsequent modules within Vis:

- View–Module

  The view module controls the overall style of representation which may vary from text to different types of graphics such as a tree diagram or a graph.

- Mapping–Module

  The mapping module connects aspect of program state to view components.

The combined effect of these two modules is to create a range of "mappings from program to pictures" (cf Roman) that can be moved amongst by the visualisation user.

Figure 4-3, taken from the Vis paper, makes this architecture clearer and includes the navigator module through which the user can control the visualisation.

PVML very precisely maps to the communication between the Domingue 'view' and 'history' modules, with PVML statements transmitting history data and a reverse flow of filtering commands being necessary to mitigate excessive volumes of program state information.

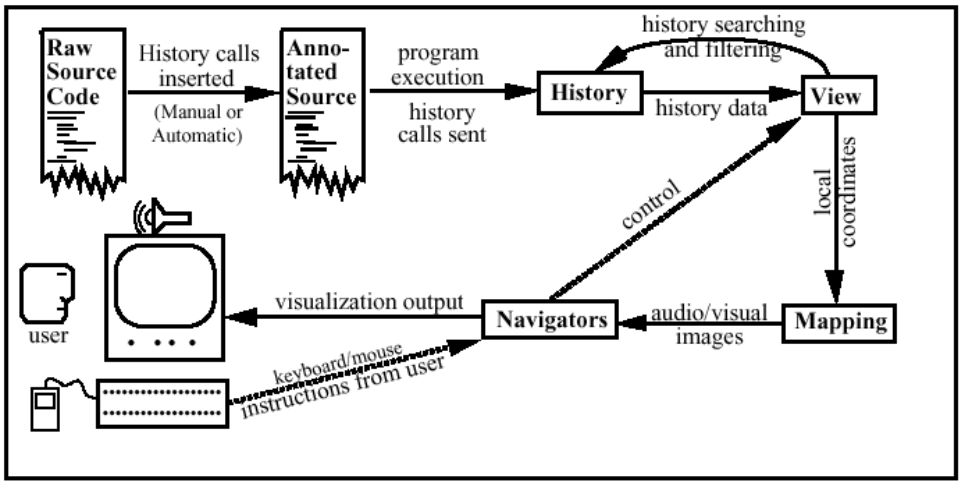

Figure 4-3: The Vis Architecture. Reproduced from [21]

4.7 Other Declarative Approaches

Two other pieces of work are referred to by Roman as to some extent making use of a declarative approach.

ALADDIN [40] was developed by Helttula et al. ALADDIN was designed to visualise Modula-2 programs and divided the issue of generating a display into a space and time axis. The question of what to display in space was handled declaratively by defining a set of *graphical types* and *graphical variables* within an animation editor, ESA, where the graphical components are associated with program states. The timing issue is handled by direct annotation of appropriate (interesting) events in the Modula-2 program by adding *ghost variables* to the Modula-2 program. These ghost variables represent program state to the visualisation and their placement determines timing.

The fact that program state is represented independently by the settings of the ghost variables and that visual representations from a library are

associated with these states is what identifies ALADDIN as a declarative approach.

The ANIMUS system [22], developed by Duisberg has been mentioned earlier in the PV review. ANIMUS achieves 'automated annotation' by the extension of class behaviour to include visual behaviour. Roman has characterised ANIMUS as declarative, based on the automatic association of objects with their visual behaviour and on the way in which *constraints* can be defined that limit the visual outcomes in selected ways.

In the case of generating visualisations based on class behaviour, the declarative label seems to be somewhat forced. ANIMUS delegates responsibility for the display of an object to the object itself and it is only to the extent that the algorithms for display of various objects are managed in an organised manner that this can be considered a declarative system.

A declarative system sets out to present a high-level, algorithmic description of program state and map that to visual states. The nebulous relationship between the values of program variables and this, more abstract, state applies just as much in an object oriented language – objects and their states are substituted for the values of variables. A coherent declarative approach will depend upon techniques to abstract the states of an arbitrary set of objects according to criteria that are driven by visualisation requirements.

Duisberg's use of constraints to manage temporal issues in the animation is of considerable interest though in refining the concept of the declarative approach. A constraint is "a statement of a relationship that we would like to have hold" at some point in the future. A constraint exists independently of the flow of control in a target program. The writer of a target program is not called upon to "write and invoke procedures to do the maintenance (of the constraint)" [9] – the constraints will be maintained by an external agent. For example a user might want to limit the number of branches in a tree diagram to the number that can be fitted on a single screen. The visualisation will be

free to run and generate arbitrary tree diagrams but the display-related constraint will be applied by another agent.

Such statements have an extensive part to play in the design of useable visualisations. The mapping of program state to visual display establishes the basic vocabulary of a visualisation but the clarity and comprehensibility of the display depends on other issues such as the relative timing and screen position of the artifacts. A perfectly reasonable representation can be rendered incomprehensible if it is obscured by other objects or it displayed with inappropriate timing. It has been demonstrated by Duisberg and others [8] [70] that a constraint-based approach is ideal for managing issues that are loosely coupled to an underlying formalism such as the display semantics of a visualisation.

4.8 Summary

The proposal for a fundamental decoupling of visualisation target and engine rests heavily on the definition of visualisation offered by Roman – best summarised in [89]. It is also supported by the equivalent, though less formally specified, work of Domingue which describes Vis, a system where the decoupling is quite explicit. The declarative approach to visualisation has been contrasted with other specification techniques and it has been shown how it offers clearer distinctions between the domains of the four principal roles involved in a visualisation scenario. This leads to the identification of the stream of program state information that must be provided as the input to a system that defines mappings between program state and pictures.

Figure 4-4 represents a Roman-like division of a complete PV system into distinct modules and locates PVML as a communication amongst those modules. The cartoons represent the three visualisation roles, user, programmer and visualiser. Each role is shown connected to a single component of the PV architecture. The emphasis on decoupling the target and engine leaves each role with a single point of contact with the system

bearing in mind, as previously emphasised, that in many cases a single human may enact more than one role.

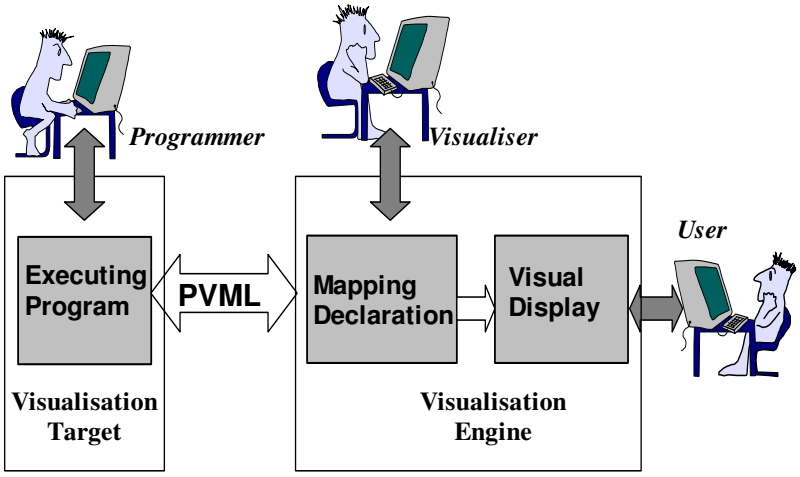

Figure 4-4: The Location of PVML

PVML is used to implement a generalised communication between visualisation targets and engines. The stream of PVML statements to the engine, represents program state information and contains no assumptions about how the program is to be visualised. These decisions are to be made further down the track, by one or more visualisers that are in control of components that are configured to consume the PVML stream. Appropriate buffering and manipulation of a PVML stream should be capable of transforming the program state information into the format required by an arbitrary visualisation engine.

The broad outline of this proposal is the subject of the author's publication [107].

Chapter 5

DEBUGGERS

In the very broadest sense the requirements for PVML have been clarified through the discussion of decoupling program visualisation systems. It is fundamental to maintaining the separation of concerns for the visualisation roles, that PVML communicate only program state information. There is no mention in PVML, of any program visualisation related data. PV declarations and manipulations are all local to the visualisation engine.

At this stage it must be noted that the language that is being defined has no inherent link to program visualisation – other than the motivation, of providing generalised decoupling amongst visualisation targets and engines. PVML is, in fact, a language that enables a selection of normal debugger functionality to be applied to a remote target program in a manner that is, wherever possible, programming language neutral.

There are numerous precedents for basing visualisation on integration with a debugger – Lens [65], Amethyst [69] as well as the large selection of commercial CASE (Computer Aided Software Engineering) tools. The objections raised to this approach in the PV literature have largely centred on the excessive level of detail revealed and the difficulty for a novice user of selecting an appropriate granularity of display. A PVML-based implementation delegates such matters to the visualiser who is in control of defining mappings between program state and visual representation.

In order to clarify the significance of PVML, as the detailed requirements are set out, they will be located within the literature that describes remote and heterogeneous debugging. The name that has been adopted for this language, "Program Visualisation Meta Language", needs to be seen as an

expression of the motivation and background to this proposal rather than an attempt to define the precise functionality of PVML.

In discussing the requirements for PVML, a first step is to locate the language more concretely in an architecture for decoupled visualisation. This is followed by relating this endeavour to other work in the field of debugging. The relationship of the language to the two main areas of concern, target program source code and target program data, is discussed. Also consideration is given to a number of ancillary areas that do not clearly fall within the scope of source and data

5.1 PVML Architecture

The intent of the PVML proposal is to interface with components of existing PV systems. Consequently the implementation of PVML needs to be in the form of drivers that interact with components from existing systems.

A driver is needed for the visualisation target – a "PVML Target Driver". This driver will be wrapped around an existing environment that supports stepping through and examining the state of a target program. Typically this will be a debugger for the programming language involved.

The second driver is needed for the visualisation engine – a "PVML Engine Driver". This driver could be wrapped around the visualisation component of an existing, or newly created, PV system. If the PV system uses the declarative approach, then the mapping declaration module will receive the appropriately formatted output of the driver. Imperative PV systems will need the mapping declaration to be implemented within the engine driver.

The two drivers interact through a two-way stream of PVML commands. The diagrams below shows the PVML target and engine drivers and the details of the interactions that they need to have with the components of a PV system and with each other.

The focus in this chapter is on the PVML target driver and the way in which it encapsulates, and abstracts, the underlying debugger.

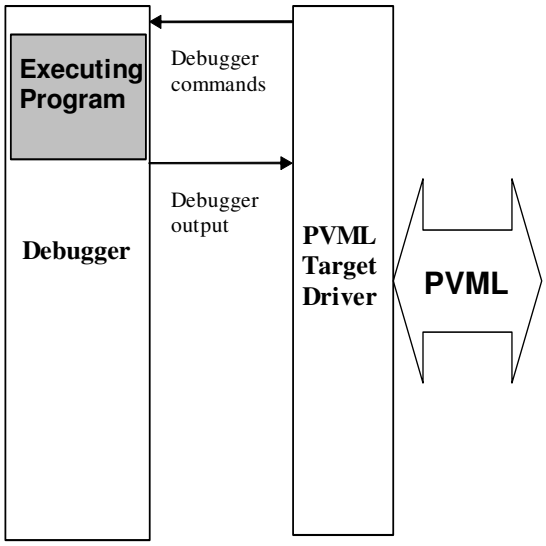

Figure 5-1: A PVML Target Driver

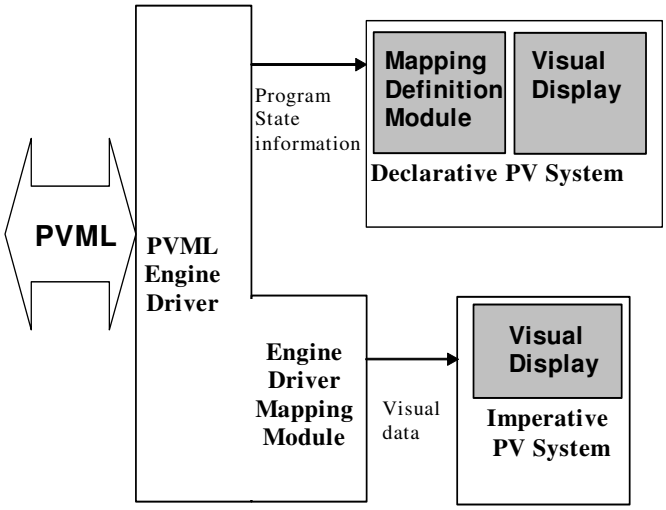

Figure 5-2: A PVML Visualisation Engine Driver and its connection to two different styles of PV display

5.2 Debuggers

In reading the extensive literature relating to debugging programs it must first be emphasised that a particular, restricted, area of the field is under consideration. The PVML approach to debugging can characterised in the following ways:

- Symbolic

  Debuggers have access to extremely low level information concerning the program that they are debugging. In many instances the programmer using a debugger will need to make use of specific memory addresses and machine implementation details to achieve the results they desire. A distinct class of debuggers makes use of the representation of the program that exists in the program source code. These debuggers are termed *symbolic* debuggers and the requirement to support novice programmer activity clearly leads to PVML supporting symbolic debugging.

  Although the novice user may well be principally interacting with visual representations of their program, it is fundamental to the endeavour of learning programming, that they will also be paying attention to the source code listing of the program they have written. It is at this point that the requirement for symbolic, rather than lower-level machine, access arises.

- Heterogeneous

  Historically a particular debugger has often been associated with working in a particular programming language as with, for example, [3], [20] and [25]. The chapter supporting the novice programming environment concept has made it clear, that for novices, the ability to apply the same programming environment to more than one language may well be useful. This leads to the requirement that PVML support what might be termed *language neutral* debugging. In the field of debuggers the term *heterogeneous* is often used to describe this ability.

  The GDB [99] debugger is a notable exception to the one-to-one debugger-to-language mapping since it supports the cross-section of languages for which there are Open Source, GNU Compiler Collection (GCC) [27], compilers available. The linking concept here is the binary executable file format and the way in which the source code information is stored – namely the program symbol table. The GDB debugger can interpret the range of executable file and symbol file formats generated by a number of GCC compilers and hence permit debugging of programs written in C, C++, Java, FORTRAN and Ada. From the point of view of the PVML proposal though, this multi-language ability cannot be considered sufficient heterogeneity since it would restrict the scope of PVML target languages to those supported by GDB.

- Sequential

  A major research preoccupation in the field of debuggers has been the issue of debugging concurrent programs [60]. This issue exposes substantial theoretical issues which are beyond the scope of this thesis. At the novice level it is reasonable to assume that students are engaging with programs that have a single thread of execution. It is specifically *sequential* debuggers that are of interest.

- Remote

  It is fundamental to the visualisation architecture proposed that the connection between the target and the engine potentially be through a network. The reasoning behind this assumption originates in the notion of the novice programming environment being location independent but it could also be argued from the point of view of maximising the extent of decoupling between the target and the engine.

  Many debuggers support this mode of operation but those that do not can be, quite reasonably, ignored.

At this stage the literature describing debuggers will be examined, restricting the view to those that are symbolic, remote-capable, sequential and language neutral.

It has been noted by Olsson [78] that debugger research has been disproportionately influenced towards the problems of debugging concurrent programs and also the provision of graphical interfaces for debuggers. A consequence of this bias is that there are comparatively few significant contributions in the restricted field of debugging that must be examined. To begin with the principal attribute that will guide the examination of debuggers will be the question of language neutrality.

In setting out to create debuggers that operate at a level of abstraction above that of a particular programming language, a predominant approach has been to define *debugging languages* that are super sets, in some sense, of the programming languages that they set out to support. Some aspects of this endeavour have issues in common with that of creating *translators* that automatically transform a program from one source language to another.

5.3 Debugging Languages

A debugging language can be characterised as providing some form of high-level abstraction of debugging primitives that exist in one or more, language specific, debugging environments. There will, in general, exist an environment with which the programmer interacts. This is the environment in which debugging language statements are manipulated and will incorporate one or more underlying, back-end debuggers that are able to host target programs in a variety of languages.

A debugging language will need to concern itself with a bidirectional flow between the programmer and the underlying debugger. Debugger commands, that trigger execution in the target program or perform specific debugging primitives, such as the setting of a breakpoint, must be sent to the debugger. At the same time, the output of the debugger must be observed and manipulated.

Although debugging languages have been described since the early days of computing science [35], a generally accepted means to classify them has not evolved. In marked contrast to the domain of program visualisation, in which there exist a significant number of well established taxonomies, debugging languages are generally classified on an ad hoc basis.

The work of Sosic [96], which does not set out to be taxonomic, in fact provides a useful axis along which to categorise debugging languages. In his

paper describing Guard, a *relative* debugger that seeks to compare the execution of two programs, Sosic divides the features of his debugger into the categories of *imperative* and *declarative*. These are exactly the same terms that were used by Roman in his classification of visualisation systems and indeed, there are many similarities in the two endeavours. The terms imperative and declarative can be applied to debugging languages in general and PVML can be located upon this axis.

5.3.1 *Imperative debugging languages*

In the visualisation domain an imperative visualisation technique was defined as one that set out to quite directly control visual outcomes. A single visualisation command, which might for example arise from annotation of the program source, would give rise to a single visual consequence.

In terms of debugging an imperative language establishes a similar one-to-one relationship between debugging language statements and the commands that are implemented by the underlying debugger. Through an imperative debugging language the programmer has, in effect, manual control of the underlying debugger – an individual command will be sent to the debugger and the output generated as a result of that command will be handled. As Sosic notes, this fine grained control may not be appropriate or manageable where the target program is complex in its behaviour. This intractability motivates the declarative model described below, specifically as a means to manage more complex debugging scenarios.

However any debugging language must contain significant imperative features in order to generate debugging commands. Using a language to generate these commands has the advantage that a protracted sequence of commands, that would be tedious to enter manually into a debugger, can be straightforwardly generated.

A typical scenario might be the examination of a target program representation of a linked list – a common data structure in which some representation stored in each element of a collection leads to the next element. Manipulating a linked list through a debugger can involve a complex sequence of commands and several authors [78], [32] have discussed the issue of generating the low level debugging commands that would be required to traverse a linked list. It is observed that when using the command set of the underlying debugger directly such an operation is tedious.

Some debuggers provide the ability to store and invoke such sequences [99] but the sequence is very specific and not easily modified. By contrast a debugging language, by offering more abstract primitives that map to a series of low level commands, make such complex data probing a routine and manageable affair.

DUEL [32] is a "very high level debugging language" that uses a syntax based on C to control and manipulate the output of GDB in a manner that is almost LISP-like.

The design of an imperative debugging language is a nontrivial issue, which will be discussed in considerable detail below. An examination is presented of some of the semantic issues involved, as described in [18] and [48].

5.3.2 Declarative debugging languages

Initially the significance of this term within visualisation will be restated. A declarative visualisation technique allows for a visual consequence to be defined as contingent upon an arbitrarily complex set of conditions that might occur in the target program. Declarative visualisation moves beyond the one-to-one mapping between program events and visual occurrences by defining a language in which complex mappings can be described.

In the domain of debugging there exists a very similar requirement. The executing program is arbitrarily complex and a number of distinct events, in

different sections of the program, may be required to trigger some debugging activity. The interesting event, from the point of view of the person debugging the program, may well have many distinct components. It is fundamental to this approach that the low level components that are combined will occur at distinct times during program execution. A declarative debugging language will contain structures through which such asynchronous events can be abstracted.

Many modern debugging languages offer an extensive set of declarative features. Dalek [78] provides debugging "at a high level of abstraction" by controlling GDB through higher level constructs. Dalek encapsulates the output of GDB into an *event* structure that supports the hierarchical processing of complex sets of events and hence the detection of arbitrarily complex states. Guard [96] provides declarative debugging through its implementation of an *assert* and *verify* command. In the domain of lazy functional languages, where execution order is non-deterministic, imperative debugging becomes impossible and the obligatory declarative approach is implemented in a language such as EDT [97]

5.3.3 PVML as a debugging language

The motivation for PVML, namely the decoupling of visualisation system architecture, requires that the debugging language operate as a more or less transparent pipe between the visualisation engine and the target program. In such an architecture the importance of the role of a visualiser, in employing a declarative approach to defining mappings between programs and pictures, has been emphasised.

Although conceptually similar to declarative debugging, the declarative aspect in this instance is at the level of mappings from program state to visual artifact rather than between a low and high level representation of program state. In order for the declarative visualisation mapping to be effective, and in order to support potential imperative visualisation engines,

the engine requires access to the kind of low-level debugging primitives and information that are represented in an imperative debugging language. Hence PVML could be characterised as an imperative debugging language and the requirements of PVML can be analysed by examining the literature describing the semantics of imperative debugging languages.

The delegation of debugger interaction to a simplified language that abstracts lower-level debugger behaviour is also the approach taken by deet [38]. The cycle of development that led from ldb [84] a "retargetable debugger", through cdb [39] a "machine independent debugger" to deet, very clearly approaches remote, language independent debugging in a similar manner to PVML. The minimal set of debugging language primitives and the implementation of these primitives in a *nub* that wraps around an established debugger represent an architecture that is similar to that of the PVML drivers. In addition, the argument that a reduced subset of generally accepted debugger functionality is an acceptable trade-off for increased portability is strongly reminiscent of the case that has been put in Chapter 2 for a novice programming environment.

The simplified, portable, command set of the PVML-based debugger can perhaps, most usefully, be characterised as implementing an 'abstract debugger', that is mapped through a target driver to a particular concrete debugger. This characterisation of PVML, as providing an abstract debugger, is one that will be used throughout the remaining chapters of this thesis.

Chapter 6

PVML LANGUAGE REQUIREMENTS

Examination of the requirements, in general terms, for an imperative debugging language leads to a definition of the specific requirements for PVML. At the highest level, it has previously been noted that a debugging language manages bidirectional communication between a user and an underlying debugger. Commands that control debugger behaviour flow from the user; and debugger output, in the form of descriptions of the state of the program, flows in the opposite direction. The requirements of the command stream are well analysed in [18] and the discussion of the reverse flow will draw upon the work of Johnson [48].

6.1 Control

In discussing the semantics of an imperative debugging language there are two fundamental concepts that must first be clarified – the question of dual or single process debugging; and the definition of the underlying debugging primitives that can be assumed to exist.

In general when debugging a program in a compiled language the debugger will run in a separate process from the target program. The assumption is that the machine architecture, on which the program is executing, provides the means for the debugger process and the target process to communicate at appropriate times.

An example of this would be the existence of a machine language instruction that generates an interrupt that can cause a context switch. A debugger, such as GDB, will insert this instruction into the target execution module at the point where a *breakpoint* has been defined. The fact that this instruction can cause a context switch means that the target program can be allowed to execute at normal speed, with the knowledge that when the breakpoint has

been reached, control will be transferred to the debugger for appropriate action to ensue.

In the case of a program written in an interpreted programming language, the debugging functionality is likely to be part of the interpreter itself, with the consequence that debugging and normal program activity take place within the same process.

Crawford [18] argues that the significance of this distinction is one of implementation efficiency, when considering the number of context switches that must occur during debugging. However, where PVML is concerned, these distinctions will be hidden within the visualisation target and the PVML stream will be unaware of whether the target debugs in a single or dual process mode.

A debugging language will be built upon certain assumptions as to the debugging primitives available in the underlying debugger. Crawford, in designing his General purpose Debugging Language (GDL) [18] assumes the existence of four primitives:

- `read`

  Read the contents of memory in the target program

- `write`

  Write the contents of memory in the target program

- `stepi`

  Cause the target program to execute a single machine instruction, as distinct from source code line (see `step` instruction below)

- `break`

  Set a breakpoint in the target program at which control will be returned to the debugger.

Crawford argues that all other, normally expected, debugging behaviour can be derived from these primitives. However, generating more sophisticated behaviour through application of these primitives has the potential to significantly add to the computation required. In the context of debuggers currently available, which potentially could be targets for PVML, the addition of four further primitives is suggested:

- `step`

  Cause the program to advance by precisely one line of source code. The definition of what represents a single line of source code will be programming language dependent. The effect of this command could be produced through iterations of the `stepi` command but it is reasonable to expect that the debuggers used will have a native implementation of `step`. The result is a considerable saving of computation.

- `resume`

  Cause the target program to resume execution at normal speed. In the absence of any breakpoints this would lead potentially to the program terminating. The effect of this primitive can be reproduced by iteratively applying the `stepi` primitive but this approach is computationally expensive.

- `watch`

  Set a watchpoint on a variable in the target program, to automatically switch control to the debugger when the marked variable is accessed. Generally a watchpoint can be configured to be sensitive to either the reading or writing of the data value.

Again Crawford describes the implementation of data watchpoints through programmatic application of the basic primitives. He suggests inspecting the value of a watched variable after each `stepi` operation. Given the support for watchpoints in current debuggers, much computation can be saved by the assumption of the `watch` primitive.

- `frame`

  Provide some representation of the current depth of nesting within successive program contexts. In a block structured language this is commonly referred to as the *stack frame*.

Crawford's GDL proceeds to generalise debugging functionality by providing the means to iterate and test the use of the primitives that he has defined. It provides the necessary looping constructs and the ability to define functions that incorporate loops constructed from a sequence of low-level accesses. By this means he is able to offer highly abstracted control of debugging functionality.

The limitation with Crawford's approach, as should be clear from the examples provided, is that working with such low-level primitives there will often have extreme performance penalties. As a result, higher level support for these instructions is desirable for practical debugging.

In the case of PVML, the language will mostly be used in communication across a network rather than the inter-process communication for which GDL is designed. In the case of this more widespread distribution it would be inappropriate, in terms of the network traffic generated, for low-level looping to be expressed in PVML. Commands in PVML must map to the upper level of a language like GDL, with any low-level looping being implemented within the target driver.

This will become clearer through the detailed discussion below of the semantics of the PVML step command.

6.2 The Step Command – a Debugging Language Scenario

At this stage, two distinct primitive implementations of a command that steps program execution have been defined. On the one hand, the `stepi` command moves forward by one machine instruction. The `step` command however, advances by one complete source code line. These are the primitive levels of execution stepping that are assumed to be provided by debuggers that PVML targets.

From the point of view of defining the requirements of the PVML language, and especially considering its usefulness to programming novices, it is quite reasonable to consider other granularities of stepping. In [18] Crawford considers stepping forwards in the source code by individual expressions, statements or even blocks of text. . In addition one might wish to support stepping into, out of and through a subroutine. These navigational devices are illustrated in Figure 6-1.

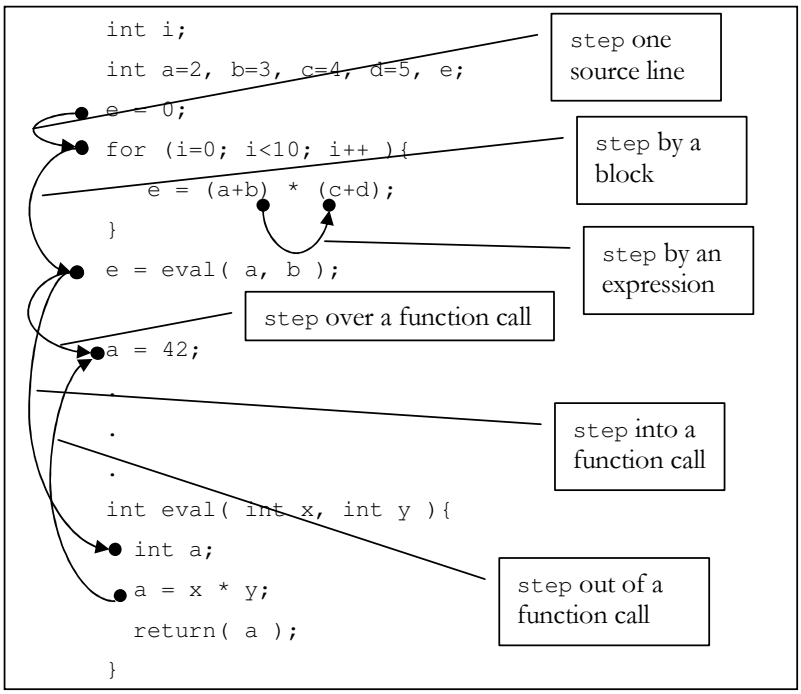

Figure 6-1 Program stepping - various granularities

Clearly, any one of these can be implemented by appropriate iterations of the `stepi` primitive, assuming that the debugger has access to appropriate mappings between source and machine code. This issue is characterised by Johnson in [49] as that of "mapping to the source language state" and the availability of the various mappings will depend on the data structures maintained by the compiler or interpreter that the debugger is interacting with.

An interpreter functions by establishing data structures that represent the program structure during execution. Run time access to these structures can yield detailed mappings between program source code and the machine code that is executed. By contrast, a compiler must analyse and translate an entire program source code into its machine code form. Intermediate data structures, that could yield an appropriate mapping, may or may not be

preserved. These structures may not even be accessible outside of the compiler itself. In general it is expected that a wider range of mappings will be available where the underlying execution takes place in an interpreter.

Using PVML to provide a high level abstraction, such as stepping by source code expressions, involves generation of an appropriate sequence of debugging primitives within the target driver. Each debugger targeted would have its own driver that would incorporate mappings from a standardised PVML form into the correct sequence of primitives.

This discussion draws attention to the issue of providing support for the differing capabilities of available debuggers and execution environments. PVML will, as it is applied to a wide range of targets, encounter disparities in target functionality and it therefore must incorporate the notion of several 'levels' of functionality.

In this case one such level is proposed for the case where only source line stepping is available, with a more sophisticated level, in which finer grained source code stepping can be provided, being applied to environments that support this.

6.3 Programming Language Issues

The flow of information from target to engine is considerably richer and more dense than the command flow that has so far been discussed. This flow clearly relates, in a much more complex fashion, to the programming language in which the target is written and also must take account of the distinction between program source code and program data. In the case of some programming languages this distinction can be complex.

In discussing programming languages, authors seek to categorise them using terms that establish a taxonomy at the highest possible level. One such term is that of the programming language *paradigm* [120]. In the scope of this

research, it is appropriate to consider three paradigms of programming language:

- Procedural languages such as Pascal or C. These are also referred to as declarative languages
- Object oriented languages such as Java or Smalltalk
- Functional languages such as Lisp or Prolog. These are also referred to as applicative languages

The question of debugging programs written in such disparate programming languages is one that requires some considerable attention. The management of program source and data could need to be adapted to suit the underlying paradigm of the programming language itself. As has been indicated previously the approach generally taken to this problem is to define a high-level, abstract debugging language – as has been done in this thesis. Automatic source-to-source translators, that accomplish their task through abstraction and reimplementation [119], must also address these issues but their approach, as a consequence of the need to regenerate source code, needs to be far more rigorous.

The work of Johnson [48] considers the question of whether some treatment of programming languages can be considered *generic*. A generic feature is not necessarily common across all languages in the sense of being absolutely fundamental. However, these features may be identified as common across several languages, including those from distinct paradigms. This is much as could be expected, considering that most languages are synthesised in a derivative fashion. Given a set of generic means to debug programs in disparate languages it is also appropriate to consider aspects that are *specific* to a language of class of languages.

Through applying a uniform debugging language, DiSpeL, to a representative selection of languages, Algol, FORTRAN, LISP and Snobol, Johnson identifies generic aspects of programs that map to what has so far been

referred to as code and data. These he terms *segment-generic* and *data-generic* features respectively. Within these categories, the specific *entities* that occur vary from language to language.

For example LISP is characterised by the code entities 'form' and 'function' whereas in the procedural language Algol, he identifies 'process', 'program', 'routine', 'clause' and 'unit'. DiSpeL debugging programs are written in terms of generic features – such as *PROCEDURE*, *BLOCK* and *STATEMENT* – and these interact with the source code of a particular program through a set of generic-to-specific mappings that relate to the source language of that program.

The core of PVML consists of terms that refer to such generic programming language features. Mapping these generic terms to more specific, language related terms, will occur in a particular target driver as required. A minimal, generalised, engine offers the visualiser access to the state of a target program through these generic terms, which the visualiser is free to interpret in ways that are appropriate to features of the source language. These activities of the visualiser role are quite independent of the PVML stream.

There may well be aspects of execution state or content, for a program in a specific language, that are not amenable to such generic treatment. A specific instance will be considered in more detail below in Section 6.5. In order to manage these language specific issues PVML incorporates a means to describe the capabilities of a particular target or engine. Upon initially establishing contact, a target and engine negotiate the extent to which their capabilities overlap and therefore what level of functionality can be provided.

6.4 Generic Code Issues

This discussion of the language requirements that relate to the management and display of program source code begins with issues that are generic to the

language paradigms under consideration. The treatment of language specific issues, that cannot be considered generic, follows as a separate section.

In considering the generic representation of program code required by a visualisation system there are two aspects of essential representation:

- Position in Source

  Representing the position of the current execution point in the source code

- Layout of Source

  Presenting the source code to the user in a manner that exposes the structure of the code (so-called 'pretty printing')

Each of these will be examined in some detail below.

6.4.1 Position in Source

Program source code is most commonly generated in a textual form. In the visual programming domain, an alternative representation is introduced, in the form of graphics.

As a program is executed a class of visualisation user, who is also enacting the programmer role, will expect the visualisation engine to show the point of current execution in the program source code. PVML needs to be able to indicate the current point of execution for programs written in a variety of languages.

There are two aspects of displaying the current position in the program source:

- Logical Line Numbers

  Establishing a consistent mapping from the debugger's notion of the current execution line to the representation of that line that is seen by the user.

- Representation of Context Change

  Adjusting the view of the source code to represent entry to, and exit from, alternate execution scopes. This aspect is treated here from a point of view that supports the broadest possible range of program language paradigms. The assumption made is that, at some level, program execution enters and leaves contexts that will be represented visually through a device such as separate source code windows or a shifting visual emphasis.

Logical Line Numbers

In a textual language it is universal to describe locations within programs in terms of the name of the source code file and the line count within that file. This is the representation used in a variety of existing visualisation systems. This approach must be mitigated, as has been noted by Mukherjea in his description of the Lens system [65], when a single statement of source code spreads over several lines. In such cases the logical line number within the program more truly represents the current point of execution. Given that the target driver is built around an underlying debugger, logical line numbers are likely to be available.

This leads to the issue of translation from logical line numbers, which are generated by the debugger, to physical line numbers that are required in order to display the source code line to the user. The assumption that underlies this question is that simply transmitting the text of every source line when it is executed would generate excessive traffic. Although the number of times a given source code line is executed is data driven and potentially unbounded the number of actual lines of source code is bounded

(though possibly large) for a given program. It seems reasonable to suggest that PVML transmit the text of a source code line the first time it is executed, along with its logical line number in order that subsequent executions of that particular line could be indicated simply by the logical line number.

In a professional development environment these issues would be complicated by the need to expand certain elements of program source code such that one physical line, as written by the programmer, may map to many logical lines, as executed. Examples of such elements are program 'macros' and in C++, 'templates'. The expansion of such elements is generally handled transparently by the compiler, but options exist through which compilers can be directed to preserve intermediate states such as macro expansion.

Whilst the initial needs of a programming novice might be met by a one-to-one physical-to-logical source line mapping, a PVML that supported more advanced programming would need to identify source lines, not simply by a digit, but through a tuple that identified a physical line and a possible logical offset within that line.

Representation of Context Change

It is common in PV systems, supporting a cross section of program language paradigms, to offer the programmer a view of source code that represents successive program contexts through a distinct visual metaphor. In the visualisation of a procedural language [14], Brown uses a separate source code window to display the code in successive Pascal procedure calls. In a functional language visualisation [46], a shifting frame is drawn around the current function in a program.

What these scenarios have in common is a means for the debug target to announce when it enters or leaves a context. This feature is not one that can

be considered a normal primitive in a cross-section of debuggers. The debugger that is part of the Java Development Kit, `jdb`, provides this feature but GDB does not. It is, however, feasible to construct this behaviour through use of other debugging primitives, in particular the combined use of the `step` and `frame` primitives. This sequence is generated within the target driver.

6.4.2 Layout of Source

The previous section has addressed the question of using PVML to relay the current execution point to the visualisation engine in order that the programmer can keep track of the program execution. The normal way in which this information is presented visually is for the current line to be displayed, with a highlight, on a view of the source code for the module being executed. Many PV systems [42], [17] make sure that the way in which the source code is displayed supports the mental models of the programmer by typographically displaying the code according to various conventions that have been shown [5] to aid program comprehension.

Examples of these conventions [Figure 6-2] are the use of different colours of text to denote programming language keywords, user-defined strings, constants and other syntactic elements, along with indenting of source code lines to correspond with the block structure of the program. The level of analysis of the target source code that is required to accomplish such a display requires access to the parse tree of the program, an intermediate, hierarchical representation of the program structure that is generated by the compiler or interpreter.

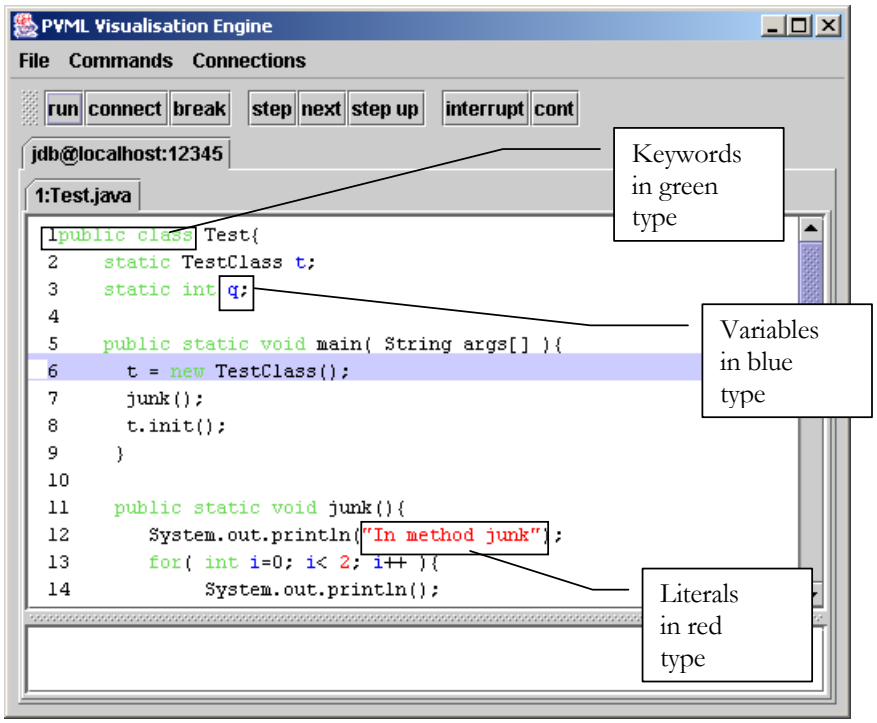

Figure 6-2 Pretty Printed Source Code

In a PVML-driven visualisation environment the visualisation engine will not necessarily have access to the parse tree of the program. The pedagogically effective presentation of the source code in the engine could be accomplished by giving the engine access to a generalised format of the parse tree, as with the intermediate representation in UWPI [42]. The assumption here is that the target has access at runtime to appropriate representations of the source program – in general it is possible to configure compilers to preserve intermediate representations, such as parse trees, that are otherwise abandoned during compilation. When this is not possible targets may explicitly re-parse the program in order to generate an appropriate representation.

This introduces the important requirement that PVML handle structured, hierarchical information such as a program parse tree.

6.5 Language-Specific Code Issues

Specific requirements for each language paradigm will be discussed.

6.5.1 Procedural Languages

The view of execution that has already been described – a highlighted line of source code is the normally accepted way to display execution in a procedural language.

An additional type of display that is often found in debuggers and CASE tools is a "call tracer" which indicates the call chain (function A called function B, which called function C) that leads to the current execution point. To support this type of display in a PVML-driven engine would require the execution target to communicate function entry and exit as previously described.

6.5.2 Object Oriented Languages

Michael Kolling [57] has argued strongly that novice programmers, who are learning an object oriented language, will benefit from a development environment that takes steps to represent the program in terms which emphasise the object based nature of the program. He draws particular attention to two aspects:

- Class and Object Hierarchy Display

  Kolling recommends that the class and object inheritance hierarchy, and the usage relationships (associations, aggregation and containership), should be graphically displayed in code and data views of the program. The distinction between code and data becomes less rigid in an object oriented language since objects created to store data have the code of the object methods included therein.

- Object Test Bench

  Objects, which may be created in the course of program execution, may also be created, and tested, manually in the development environment.

These two requirements have different levels of impact on the PVML proposal and are discussed in greater detail below:

Class and Object Hierarchy Display

Some aspects of the display of class and object hierarchies can be considered simply to be a visualisation issue and therefore beyond the scope of PVML.

For example it could be a visualiser's decision to display the execution of a C (procedural) program as if it were object oriented, with functions in modules being represented as if they were methods of objects. Similarly the execution of an object oriented program could be portrayed as if it were procedural and this is indeed the objection that Kolling made to many development environments at the time. In neither case is there a need to communicate any different information at the PVML level – what is changing is the way that the program state is being mapped to pictures.

The requirement to display inheritance hierarchies at the same time though, requires an additional level of information and some significant additional considerations to be introduced into PVML. The requirement is that all or part or all of the inheritance hierarchy is communicated to the visualisation engine. This level of communication would support the generation of visualisations similar to those available in BlueJ, such as those shown in Figure 6-3

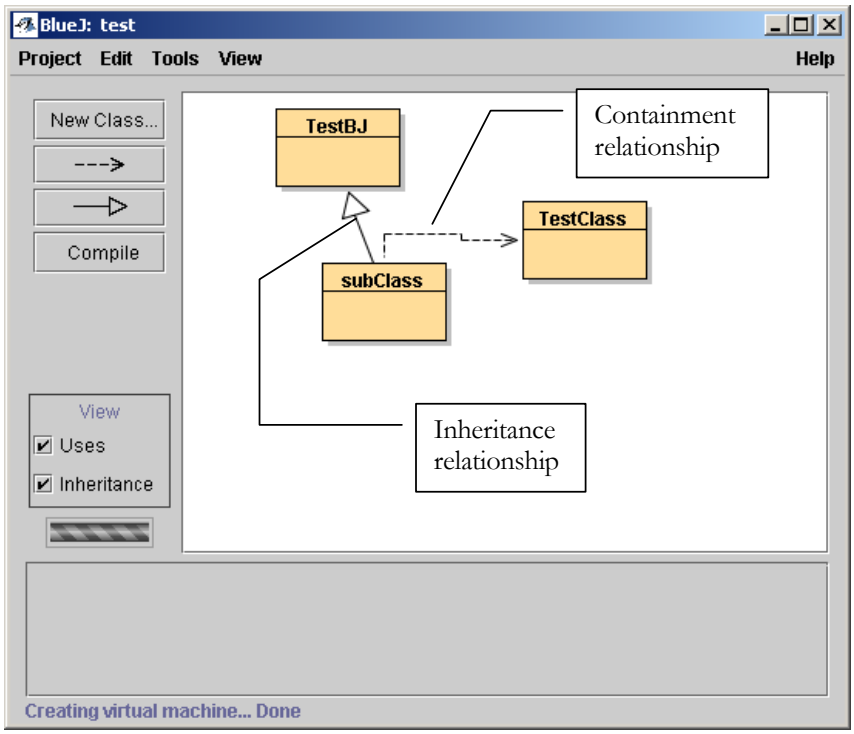

Figure 6-3 Class relationship visualisation in BlueJ

This question shares many aspects of the transmission of structured source code considered above – namely the possibly selective delivery of a large, tree structured data set to the visualisation engine. Some portions of this stream will be code and some portions data, reflecting the intermingling of code and data in the object oriented paradigm.

Object Test-Bench

An outstanding feature of the Kolling BlueJ environment [57] is the ability to place objects on a test bench and interact with them outside of the normal program code as shown in Figure 6-4.

This feature encourages development of prototype classes, the reuse of code and experimentation with objects and classes beyond the scope of a particular program. This feature is so beneficial to novice Java programmers

that it is worth considering whether such a facility could be supported by a PVML-based environment.

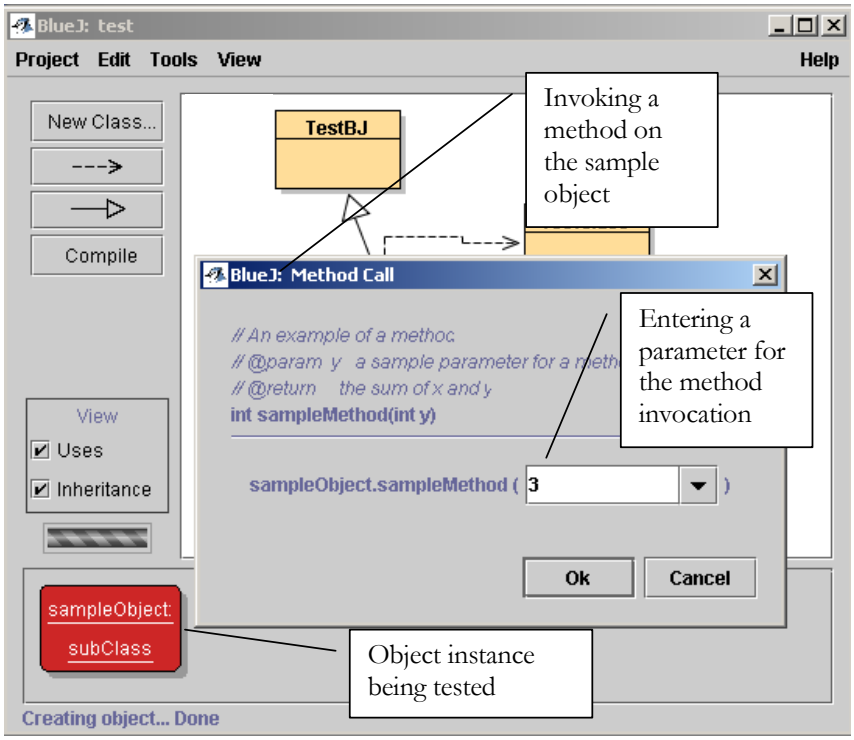

Figure 6-4 Object test-bench in BlueJ

This introduces the requirement that PVML support commands to instantiate and invoke methods upon target objects. This particular scenario will treated in more detail as an example of a programming language specific extension to PVML. In principle the visualisation of the resulting objects should be handled by the mechanisms already provided.

6.5.3 Functional Languages

Functional languages differ from procedural, and object oriented languages in their use of data values and structures. A program written in a procedural language generally makes extensive use of program variables to store data values. These data values are manipulated, and passed among, functions in

the program but the return value of a function is often of incidental interest, perhaps indicating an error condition[2].

By contrast, a functional language performs most manipulations by calling functions which themselves may have other functions as parameters. The return values of functions are primitives of the type that is germane to the language – for Lisp it is lists and for Prolog it is predicates – but the point about the functional paradigm is that the function return values are not (necessarily) assigned to a variable; they are merely passed, on the stack, from one function call to another.

The problems of visualising evaluation of programs written in functional languages have been addressed by Touretzky [110] and by Jimenez-Peris [46] and it is worth noting that in describing the execution of programs written in functional languages there is little difference between visualising the progress of execution and visualising data in the program. The sharp division between these two concepts that exists in the other programming paradigms is blurred by the fact that most of the data in the program is the return value of the functions.

The only distinctive message, in terms of program execution state, that must be relayed by PVML is the entering and leaving of functions – along with the parameters and return values. None of this is additional to what has already been covered.

---

[2] It is certainly possible to write procedural language code in a functional style – there is a place on the stack for a function return value and though the data type of the return value is limited in most languages a pointer can always be used. It is also possible, in all but the purest functional languages, to declare variables and program as if the language were procedural but in both cases one would be doing an injustice to the intent of the language.

6.6 Data

Visualisation of program data is an area that is exceptionally thoroughly covered by existing PV systems. In terms of the decoupling achieved by adopting a declarative approach to visualisation, it is the program data that defines the program state and visualisation mappings will consist of associations between combinations of data values and the pictures that represent them.

Because the intent is that it is the visualiser that makes algorithmic selections of what aspects of program state to display, the visualisation engine potentially needs to have access to an unfiltered stream of program data state information. Clearly there will be occasions where this stream is excessive and contains un-needed information but an important aspect of PVML will be a means to apply constraints to this flow of information, which is expanded upon in Section 6.7.

Another interesting consequence of feeding a PVML stream to a declarative visualisation engine is that all the algorithmic decisions will be made later – after the PVML stream has been constructed and consumed. There is no need for any algorithmic level of description to be implemented in the PVML language.

When considering the representation of target program data in PVML there are some general issues as well as considerations that are specific to particular languages and paradigms.

6.6.1 Data Values

Ultimately it is the values of data items that are of interest to the programmer and in many cases specific variables in the source language will contain the data values. The manner of this containment, which can be language dependent, is not the concern of PVML. What is significant is that the engine can provide a specific variable name (taking account of scope) and the

target can respond with the value of that variable. The variable may represent a complex entity, such as a Java object or a C structure, but the values required are contained within the representation of that variable.

What is convenient about such representations is that they are self-contained in the sense that the representation has a beginning, various intermediate hierarchical levels, and an end. The representation is bounded and well-behaved.

6.6.2 Data References

In many languages it is not necessary for all variables to 'contain' the values of data items. The alternative is that a program variable refers to, or points to, the value being stored. In C and C++ this type of reference is known as a *pointer* [55]. The programmer is still interested in seeing the value of the data that is pointed to, but the straightforward association of the program variable with the data value no longer exists.

The pointer may, on one occasion, reference a certain data item. At a latter stage of execution the same pointer may reference a different data item. Both data items may still, with complete validity, exist and need to be shown separately to the programmer but a means to reference them has to be found that lies beyond the program variable that stores the pointer.

This is a straightforward issue in a debugger that runs on the same machine as the target program. The means to refer to the two sets of data values independently is the machine address of the data item. When considering a remote configuration, such as PVML supports, it is important to realise that a machine address in the target has no meaning at the engine, other than as a value that can be passed back, at a later stage to the target. The assumption is made that memory references will be persistent during program execution. This assumption has the effect of excluding certain operating system and

debugger combinations in which virtual memory addressing is not 'hidden' from the programmer.

The target driver needs to be able to:

- Resolve ('de-reference') such pointer values, returning the referenced data value to the engine. The underlying debugger would need to be aware of the program variable semantics of pointer values – namely that a particular machine address is, in fact, a pointer to a particular structured data value. The target needs to be able to apply such dereferencing recursively, such that when the data pointed to is itself a pointer it is, in turn, de-referenced.
- Keep track of changes in the usage of memory in the target program that could cause pointer references to become invalid. The `watch` mechanism of the underlying debugger would need to be capable of keeping track of changes in data at arbitrary memory locations.
- Limit the extent to which recurrent series of pointers are followed. Following a 'pointer chain' is a potentially unbounded activity – possibly even one that repeats infinitely as when a particular pointer value leads back to the start of the chain. PVML needs to be able to specify how many steps should be taken along such chains.

6.6.3 Procedural Languages

The visualisation of data structures and values in procedural languages is perhaps the aspect that most distinctly characterises the broad area of program visualisation. The most widely cited examples of PV systems, such as BALSA [14], Tango [103], Zeus [16], all make some attempt to visualise the data structures of a procedural language. This is true irrespective of whether the work is styled as program or algorithm visualisation. A fundamental aspect of software visualisation, across the board, is some kind of representation of program data.

The representation desired, from a novice programmer perspective, needs to transparently display values in a manner that relates strongly to source language constructs and weakly, or not at all, to underlying machine implementation details. Figure 6-5 shows the view presented by DDD [29] of a linked list implemented in C. Whilst the use of C variable names (`list`, `value`, `self`, `next`) is helpful to the novice, the machine addresses (eg `0x804ab78`) are quite probably not.

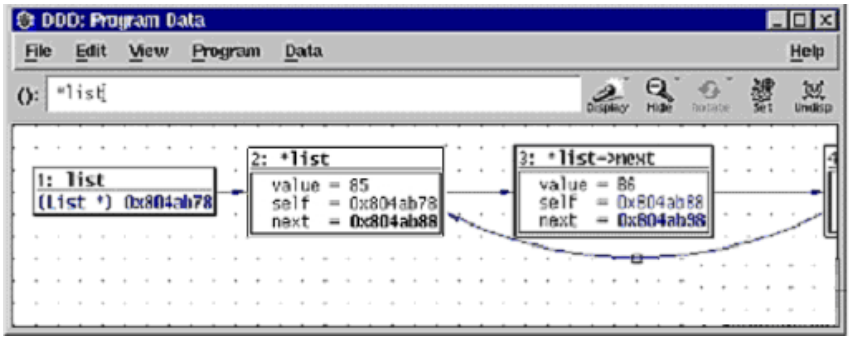

Figure 6-5 Data Structure Visualisation in DDD. Reproduced from [28]

As has been observed it is the selection of what kind of data is displayed that characterises a system as program visualisation (PV) or algorithm visualisation (AV) and the means for eliciting information about data in a running program is generally some form of annotation or instrumentation. Automatic annotation is generally associated with PV and manual annotation with AV but it is fundamental to the decoupling implied by PVML that annotation will be automatic – namely that the source program itself will not be modified to support visualisation.

It is important to note that the description of the state of arbitrary data structures implies that PVML will have a syntax which includes terms that described hierarchical data. This observation has already been made in the context of source code, as well as object hierarchy, delivery.

6.6.4 Object Oriented Languages

As has been previously noted, the distinction between program code and data becomes less sharp in an object oriented language. The earlier section that discussed the description of program execution for object oriented languages therefore has some relevance. The advantages of representing the program in terms which emphasise its object based nature have been argued by Kolling [57] as has the consequence for the PVML stream, namely that aspects of the class hierarchy of the target program are relayed to the engine.

As program execution proceeds and changed data objects are relayed to the engine, the PVML target should take steps to avoid the resending of the code component (the method implementations) of watched objects.

6.6.5 Functional Languages

Literature relating to the visualisation of functional languages has already been cited in the discussion of presentation of program execution. The relationship between program execution and data is such the observations concerning code also refer to data in the functional paradigm.

In the Lisp language the point at which values are computed is the internal 'EVAL' and 'APPLY' functions and Lisp visualisers such as the work of Touretzky [110] establish a hook into these EVAL and TRACE calls in the Lisp interpreter and generate output at these points. Touretzky's visualisations are in the form of static EVAL and TRACE diagrams generated by a set of LaTeX macros. This approach could clearly be used to yield a stream of data-related PVML statements.

Prolog execution proceeds as successive predicates are evaluated and a Prolog visualisation such as TPM [23] hooks the internal steps of the Prolog interpreter – namely the attempt to prove a goal, the successful proof of a goal, various levels of failure and re-attempting to prove a goal. It has been demonstrated by the architecture of environments such as TPM that these

steps can be hooked. A PVML driver for a Prolog target would generate an appropriate PVML stream at each of these points.

6.7 Managing Traffic Volume

The volume of traffic that passes between the target and an engine can critically affect the usability of a decoupled PV system. In a typical PVML usage scenario, target and engine will be executing on remote machines and this traffic will actually be network traffic. A requirement of PVML is that the language, or its implementation, contains features that can manage traffic volume.

The volume of traffic will depend on the extent of the display being provided and the level of detail in the visual representations. These issues are principally ones that will concern the visualiser role – the visualiser will, on the one hand, select particular data as being of interest and on the other hand wish to specify the visual interpretation of that data. This level of detail is often referred to as *granularity*.

The granularity issue is germane to all PV. Price [83] characterises this issue as *elision control* making the observation that irrelevant information may need to be suppressed and that the problem grows with the size of the project. Jeliot [36] allows the user to configure program variables to be present (or not) on a stage on which the visualisation is enacted. TPM [23] allows the user to choose between a long distance view or a close-up view of the boolean decision tree (which TPM refers to as the And/Or Tree Augmented or AORTA).

The PVML stream from an executing program will consist primarily of descriptions of regions of source code and representations of the values of data. The flow of source information, as has already been discussed, will be mitigated by caching source code at the engine. A repeated request for source code can therefore be satisfied locally, from the cache. The

representations of data are considerably more complex and raise several issues.

Through the debugger-style *watch* command, the visualiser can request information describing all aspects of the data state of the target program and is therefore in a position to define arbitrary states, made up of a combination of values that are to be visualised. The second step in creating a visualisation, following the declarative approach, is to define mappings from these program states to selected visual effects. This defines a completely functional PV system.

The decisions made by the visualiser, as they devise the mapping from program state to visual representation, results in selectiveness at two levels being applied to stream of data descriptions:

- What to View

  The visualiser, or possibly the user if the engine permits, will select certain data to be viewed. The consequence of a selection will be a request to the underlying debugger at the target to place a watch on the data member and updates will be transmitted.

- How to View

  The visualiser will also be making decisions as to what form of visual representation is presented to the user. This has a number of consequences for the language.

What to View

The selection of a data item that will be watched by the debugger in the target may be the result of a specific desire, on the part of the user or visualiser, to view that data item or, in a more sophisticated declarative scenario, that particular data item may constitute one component of a larger mapping scenario. In either case, the engine can use a PVML command to request that the target debugger watches the variable. This has much in

common with the imperative approach to program visualisation in that certain events are being designated as 'interesting'.

The volume of traffic that arises from watching a large, possibly complex, data structure at the target is potentially unbounded and it will be a necessary for PVML to provide a mechanism that can avoid the sending of data values that have not changed. If only a single component of a large structure has changed in value the target driver needs to be able to detect this and the language will be required to transmit what might be considered *deltas* of a data structure, rather than the entire structure.

How to View

The default behaviour that has been described, with regards to data, is that the portion of a data structure that has been modified be transmitted in its entirety. The design of visual representations that are appropriate for the novice programmer may well require that there is some control over the level of detail displayed to the user. As has been already stated this problem is not new to PV developers – there is a well-established case, in the PV field, for the elision, under certain circumstances, of aspects of the full view of program state. The object of this section is to assess techniques that have been used in related contexts and therefore to suggest the means by which a PVML stream could be filtered or reduced.

The Vis architecture (Figure 4-3) implements filtering of the annotation stream as a form of 'back chat' from the view module to the history module. The predominant flow is in the opposite direction – a stream of history events that may, or may not, be mapped to pictures. The visualiser may, by selections made in the view module, filter or even search through the collection of history data.

The design of PVML intersects with this implementation in several ways:

- The Vis history module needs to maintain a complete set of history entries in order to support the searching and filtering. It is not anticipated that this would be the case in PVML. This would have implications concerning the size of the target driver program. It is envisaged that the target driver would be a relatively lightweight piece of software that generated a PVML stream and delegated such issues as searching the history of events to the visualisation engine
- The storage of history records at the target in Vis enables filtering and searching to both be implemented there. The PVML design implies that searching, when supported, will be implemented in the engine driver whereas the actual filtering of the PVML stream will take place in the target driver. The implication of applying the filter here is that the traffic that has been filtered will not be generated. In a remote visualisation scenario, where target and engine are on separate hosts, the filtered traffic will simply not appear on the network.
- The specification of filter patterns in Vis takes place in the view module which equates to the visualisation engine in the PVML architecture. This is an appropriate location.

Most aspects of the PVML stream that have been discussed so far have related to the information that needs to flow from the visualisation target to the engine. It has been noted that many components of this information are hierarchical in nature. The examples that have been given include the state of target data, the target source code and the target object hierarchy. The means to define this flow of hierarchical information has yet to be specified but the requirement to filter regions within this flow very clearly has implications that have a fundamental impact on the implementation of PVML

The means of transmitting filtering requirements from the engine to the target should clearly be in harmony with the means adopted to transmit the data in the opposite direction.

6.8 Ancillary commands

The location of PVML, within the architecture of a distributed novice programming environment, dictates that aspects other that debugging must be managed. The novice programmer will be undertaking all steps in the development cycle though the engine that constitutes their development environment and so the language must include commands and responses that map to such aspects as management of student source code and, when required, compilation.

PVML will need to handle the commands to request compilation, possibly incorporating a subset of compiler options. The compiler option to support debugging of the target program would be part of the default compilation request made by the target.

The compiler errors caused by program syntax errors would need to be relayed to the engine and displayed to the user in a way that was helpful for a novice and sympathetic to the programming language being used. A comment made by Johnson in [48] is pertinent in this regard – "although the debugging system should be language-independent, it should appear language –dependent from the user's point of view". Through PVML the novice programmer would be presented with the compiler error messages and warnings that are specific to the programming language they are working with.

6.9 Summary of PVML Requirements

The considerations discussed in this chapter lead to the definition of a set of requirements for PVML which are presented here in Table 6-1 and Table 6-2. This list expresses the core of PVML that would provide access to generic visualisation of a cross section of programming languages. The table separates PVML statements into those that are sent by the engine to the target ('commands') and those that flow in the opposite direction ('replies'). The

terminology used deliberately avoids the terms 'request' and 'response' since these terms acquire a more specific meaning in Chapter 7 when an actual implementation of PVML is described.

The question of specific requirements that arise in the context of a particular language or programming paradigm is one that is open ended. A description is offered of a single language-specific scenario, that of the object test bench, in order to illustrate a general direction that might be followed by subsequent extensions to PVML.

PVML Statement	**Comment**	**Parameters**
Generic PVML Commands - Sent by the engine to the target		
break	Set a breakpoint.	Location of breakpoint
compile	Recompile program.	Compiler switches
cont	Resume normal (ie non-stepped) execution. The program will execute until it terminates or meets a break point	
file	Request a target file system listing	Identity of target location
list	Provide source listing.	Identity of region of source
next	Advance execution by one source line in the current execution context – this could involve executing an entire sub routine or function.	
query	Request capabilities of target.	
read	Read a memory region	Identity of region
run	Cause the target program to load – but not execute.	Identity of program
save	Save program text.	Identity of program and code to save
step	Advance execution by one source line in the entire program.	Optional parameter to step out of a context
stepi	Advance execution by one source expression	
watch	Set a data watch point.	Identity of data item
write	Write values to a memory region	Identity of region and value to write

Table 6-1: Generic PVML (engine to target)

PVML Statement	Comment	Parameters
Generic PVML Replies - Sent by the target to the engine		
code	Source code listing in response to 'list'.	Representation of source code
breakresp	Confirmation of 'break' command.	Success or failure
data	Data value resulting from the triggering of a watch	Representation of data value
location	Updated current execution point resulting from step/next/cont	Representation of location
pvmlinfo	Response to 'query' command.	Representation of target ability
frame	Indicates that the execution context has changed	Extent and direction of change
fileresp	Response to 'file' command	Representation of a region of target file system
error	A target error that must be communicated to the engine	Representation of target error
saveresp	Response to 'save' command	Success or failure

Table 6-2: Generic PVML (target to engine)

PVML Statement	Comment	Parameters
Specific PVML Commands Sent by the engine to the target		
Instantiate	Cause the target program instantiate an object	Identity of object class Any necessary parameters for the instantiation
invoke	Cause the target program to invoke a method on an object.	Identity of object and method Any necessary parameters
Specific PVML Responses Sent by the target to the engine		
instantiateresp	Confirms the result of an instantiate request	Failure or else identity of object
invokeresp	Result of invocation of method. Data watches may be triggered causing 'data' responses as well.	Direct output of the method invocation

Table 6-3 Specific PVML for the Object Test Bench scenario

The next chapter discusses, in general terms, the means that might be employed to implement a PVML and move on to describe an implementation that has been undertaken during this research. This implementation is capable of communicating between PVML engines and targets in a range of declarative languages.

In a subsequent chapter the application of this definition of PVML is described. A single reference implementation of a PVML engine offers rudimentary debugging access to programs hosted by a pair of reference PVML targets. The underlying debuggers in these targets are JDB, the Java debugger, and GDB, the GNU debugger. The target programming languages supported hence include Java and the set of languages supported by GDB.

Chapter 7

REFERENCE PVML IMPLEMENTATION

PVML has been described as a language that will provide communication between a visualisation engine and remote visualisation targets. Given a variety of network infrastructures through which such communication may need to occur, there could indeed be a variety of ways in which PVML was implemented. In order to demonstrate the proposed language, and to a certain extent evaluate its use, this research includes a reference implementation of the PVML language. Hence forward in this thesis, all mentions of PVML should be regarded as referring to this reference implementation. This implementation embodies the following constraints:

- Network Infrastructure

  The ground work that was undertaken in defining a novice programming environment leads to the requirement that target and engine interact through the Internet as currently configured. This means that target and engine may well be separated by arbitrary layers of Internet security mechanisms.

- User Interface Environment

  As was discussed in Chapter 2, the location portability of the programming environment is considerably enhanced by implementing the engine in a manner that supports execution in a Web browser.

- PVML Language Scope

  The reference implementation of the PVML language will be restricted to features that have been identified as generic across the three language paradigms (Section 6.5) that have been considered.

7.1 PVML Distribution Platform

In the general sense, the combination of visualisation engine and target communicating through PVML constitutes a distributed application. There is a wide range of distribution architectures that can tie together such components but the requirements of the reference language implementation, specified above, significantly limits the choice of distribution architecture. Indeed, during the time span of this research, even the term 'distribution architecture' has come to be less apposite for a number of reasons.

In [59] Matter traces the evolution of distributed systems, drawing attention to the tightness of the coupling between distributed components when the architecture of distribution is based on the notion of remote procedure call. Remote procedure call, the metaphor that lies behind a wide range of application distribution architectures such as CORBA [79], COM+ [62], RMI [45] and RPC [108] requires that a client application behaves as if the procedures implemented in the server were a part of the local program. This aspect of the implementation has two profound consequences:

- Specialised Libraries

  Significant, and specialised, communication and data-packaging libraries become part of both the client and server application. This can restrict the platforms from which components can readily be deployed. For example a Microsoft browser often does not include the libraries to support RMI or CORBA whereas other browsers might have problems with a COM+ distribution.

- Programming Language Semantics

  It is fundamental to remote procedure call that the semantics of the client application and of the services invoked in the server must match at a programming language level. Whilst many architectures (RPC, CORBA) abstract this through the use of a language-neutral Interface Definition Language (IDL) it remains the case that a procedure call is made in the client that will only be returned from when the server has completed executing that request.

The result is that there exists a very tight coupling between the client and the server – moreover one that depends significantly for its operation on the precise browser platform in use.

The alternative is for distributed components to interact through far more loosely-coupled frameworks – the approach adopted in the *Web Services* [115] the architecture currently evolving through the World Wide Web Consortium (W3C). What characterises the web services approach is the use of the standard web protocol, HTTP, and the ubiquitous data format, XML [13], to link remote components [98].

In this light, the distribution technology employed by PVML needs to be assessed with respect to communications protocol, and rendering:

- Internet Protocol Issues

  The networking infrastructure issues raised above will influence the low-level network protocol that encapsulates the PVML traffic. The aim is to maximise the likelihood that an arbitrary target, running on a secure institutional server, can communicate with an arbitrary engine that will be running in a possibly insecure location elsewhere on the Internet. This suggests that whatever form the PVML messages take they should ultimately be encapsulated in HTTP – the standard protocol of the WWW.

- Browser Implementation Issues

  The distribution architecture selected needs to be one that integrates easily with the major browser platforms in use. In practice, the ongoing market struggle between Microsoft and other suppliers means that browser support for different application distribution schemes is by no means heterogeneous.

Both of these factors, which underpin the location-independent deployment of targets and engines, lead towards the proposal that the reference version of PVML be implemented using XML. An XML definition of PVML will integrate transparently into a web services framework if that is required in the future. Apart from these deployment issues, an attractive aspect of XML is its handling of hierarchical data – a feature that is fundamental to the traffic between visualisation targets and engines. This will be the focus in the following section.

7.2 XML-based PVML

Throughout the computing industry the description of arbitrary hierarchical structures is increasingly being handled by XML [13]. Despite the origins of XML as a means to create user-defined tags within HTML documents, the fact that XML provides a "linear syntax for trees" [53], means that XML is

being used in many domains aside from the layout of Web pages. The following examples represent the breadth of such application:

- XML definition of structured document formats underpins many current open e-Commerce proposals
- The Object Management Group (OMG) who define open standards for distributed object technology (such as CORBA) have defined a class hierarchy interchange format that uses XML [80]
- XML has been used to implement incremental code migration [26]
- XML has been used to define source code profiling specifications [105]

The latter three examples all demonstrate the use of XML to describe program related constructs of a hierarchical nature, object hierarchies, program code and program execution respectively.

The description of what constitutes a legal set of XML statements in a particular context is defined by a schema-like document known as a Document Type Definition (DTD). The DTD defines the layout and legal content of an XML document which provides the extensibility of the language. New terms can be added to a document simply by defining them in the DTD that is attached to the document.

There are many precedents for defining a new language in terms of XML. The Organisation for the Advancement of Structured Information Standards (OASIS) lists more than 500 'XML Applications' [76], each of which involves the definition of a DTD. Some examples include:

- Bioinformatic Sequence Markup Language (BSML) [74]
- Taxonomic Markup Language (TML) [77]
- Chess Markup Language (chessML) [75]

The PVML DTD will be presented in Section 7.3 but some preliminary discussion will clarify some aspects of the DTD.

7.2.1 Request and Response

The top-level distinction in PVML is between a *request* and a *response*.

PVML requests arrive, either at the target or at the engine, asynchronously. This means that engine and target must be written in a way that can handle a PVML request at any time. An example of this would be a data value returned from the target, as a result of a watch that has been placed on a variable. This message will be generated by the target at a point in time that bears no consistent relationship with user activities in the engine. It is simply a side-effect of program execution. The engine needs to respond to this request with appropriate visual behaviour.

PVML responses always occur as a result of a previous request. Responses are synchronous and should be waited for. All of the defined responses flow from the target to the engine and are the result of engine requests. An example would be the engine requesting a program listing and receiving the response that is the listing.

7.2.2 Engine to Target Requests

The requests that the engine sends to the target are all straightforward commands that map to some combination of debugger primitives. These are listed below:

- `run`

  The run request begins a session with a particular execution file. The parameter to the run request consists of a file system identifier through which the target can locate the executable. The assumption is that the executable has been compiled in such a way that it can be debugged. A fully functional engine would provide a remote file system browser, driven by a sequence of PVML requests and responses, that would generate the file system identifier in response to user selections.

- `step`

  The step request causes the target to advance execution by one line of source code in the entire program. If the current execution point is a subroutine or function call then the line of source code that is executed will be the first line in that function and the current execution point will have moved to a new context. All debuggers support this basic mode of operation.

  An optional parameter to the step request will cause execution to advance until the current context terminates. In debuggers this is often referred to as 'step up'.

- `stepi`

  The stepi request causes the target to advance execution by one source code expression. This does not map to any normal debugger primitive. The stepi command that is available in many debuggers is in fact a command to advance by a single machine code instruction. Whilst appropriate in a debugger for professional programmers the novice programmer requires a granularity of stepping that corresponds with the source code entities that they are manipulating. For the target to provide this command there needs to be a mapping available between source code expressions and machine code locations. Given such a mapping a series of primitive stepi commands could be invoked on the underlying debugger by the target driver to cause a PVML stepi to take place.

- `next`

  The next request causes execution to advance by one line of source code, in the current context. If the current execution point is a subroutine or function call then that entire function will be executed. In debuggers this is often referred to as 'step over'.

- `cont`

  The cont request causes execution to advance at 'normal' speed. In a PVML context this speed of execution will be limited by processing that the target driver must undertake to implement other features, such as a generalised data watchpoint facility. Upon receiving a cont request the target will proceed to execute until a breakpoint is reached or the program terminates.

- `break`

  The break request sets or clears a breakpoint in the target program. The PVML break request maps directly to a straightforward debugger break command. There is no support for conditional breakpoints that will be triggered only when certain data values exist. The break request is accompanied by a parameter that identifies the source code location where the break is to be set or cleared. This will be expressed in terms of a source file name and source line number.

- `list`

  The list request will trigger a response from the target containing a representation of program source code. The parameter that accompanies this request identifies the source filename. Future enhancements of PVML would allow regions within a source file to be specified.

- `watch`

  The watch request sets or clears a watch on a variable in the target program. The parameters indicate whether this is a set or clear operation and identify the variable using the source file name, the procedure name and the variable name. In using this request the target and engine need to arrange that the scoping rules of the programming language are observed. Uniquely identifying a variable can raise many programming language dependent issues. Appendix A resolves this issue in greater detail.

- `query`

  The query request is the means by which the engine discovers the capabilities of the target. There could exist, in an environment supporting various language paradigms, a possibly complex range of capabilities.

- `save`

  The save request is used by the engine to request that modified source code is saved to the target file system. The parameters to this request consist of the full path name and the modified source code.

- `file`

  The file request is used by the engine to manage a file browser dialog that would enable a user to browse their file system space on the target machine. The parameter to this request consists of the path name that is to the browsed.

7.2.3 Target to Engine Requests

Asynchronous data, that must be sent from the target to the engine, will be contained in a request message. In particular this is the means by which changing data values are relayed in order that they may be visualised.

- `frame`

  The frame request describes a change in execution context that has taken place at the target. Execution context changes are handled as potentially asynchronous events since they may occur during the execution that proceeds after a `cont` request. In this instance changes in source code view may need to be displayed visually as the target program moves between execution contexts over a protracted period.

  In stepped execution, the execution of a single line of source code will result, if a frame change occurs, in the addition (push) or removal (pop) of a single frame from the program execution stack. The normal parameter passed with a frame request hence needs to be plus (or minus) one.

- `data`

  The data request is the means by which the target communicates data values to the engine. Section 6.7 has discussed ways in which the volume of this stream could be mitigated – in particular to isolate regions of a complex data structure, either because they had actually changed or to support a selected granularity level. The reference implementation of PVML contains no features of this nature. Entire data structures are sent in the form of a hierarchical description that contains variable values and types. This is discussed in greater detail in Appendix B.

7.2.4 Target to Engine Responses

The balance of PVML traffic will consist of responses that the target generates to the various requests described. Neither of the two requests that flow from target engine require any response.

- `code`

  The code response contains a representation of the program source code and replies to a list request. The representation of target source code can be at three levels of detail according to the abilities of the particular debugger and program source language. This representation is described in detail in Appendix C.

- `breakresp`

  The breakresp response is the acknowledgement of a request to set, or clear, a breakpoint. A simple success or failure code is the parameter passed with this response.

- `location`

  A location response is received by the engine as a result of any command that causes target execution to advance. The arrival of a location response is an indication that the target has successfully advanced to the location specified and an engine would be able to highlight an appropriate source line. The location is described by means of a source file name and line number parameter. It should be borne in mind that whilst a location response is pending there can be an arbitrary number of data and frame requests arriving at the engine each of which may have a visual consequence.

- `pvmlinfo`

  A `pvmlinfo` response is the reply to a query request and the parameter must communicate the capabilities of the target. The reference version of PVML does not make use of this facility. The name of the target debugger is passed as a placeholder.

- `saveresp`

  A `saveresp` response is the reply to a file request and indicates success or failure of the save operation.

- `fileresp`

  A `fileresp` response is the reply to a file request. The parameter consists of the identity and types (file or directory) of file system objects at the level that is being queried.

7.3 PVML Document Type Definition

The authoritative definition of PVML, from an XML point of view, is contained in the DTD presented in Appendix D.

There are many aspects of the format of an XML document format that are not described in a DTD. The DTD has the purpose of defining the containment rules but does not provide any support for the checking of leaf nodes. The leaf nodes are defined at the foot of the DTD and are all denoted as `(#PCDATA)` which, in terms of the automatic checking of documents, is nothing more than a commitment to include some bytes of data at that point. A lower-level validation of an XML document would be accomplished through the use of XML Schemas [118]. This degree of automated validation of the PVML stream has not been undertaken here but would, quite reasonably, be part of a wide-spread implementation of PVML.

7.4 Examples

The discussion of PVML requirements has hinted at a variety of programming scenarios to which PVML can be applied. This section presents the PVML traffic involved in a series of such scenarios. The captures of PVML traffic have taken place between reference implementations of PVML components. Chapter 8 describes the reference engine and the two targets that have been implemented in the course of this research. These packages have been configured to dump the PVML traffic and that traffic is presented here.

7.4.1 Loading a Java Program

This shows the engine requesting the loading of a Java program and the transmission of the source code. The result will be a source display as shown in Figure 6-2 on page 87. Large portions of the PVML traffic have been removed in order to focus on significant aspects.

Figure 7-1 shows the initial request from the engine for the target to run a program. Figure 7-2 shows the beginning of the resulting response that sends the source code. When the source code response is complete the target will send (Figure 7-3) a frame request to trigger an initial stack frame representation followed by the position response that will result in the initial source line being highlighted (Figure 6-2)

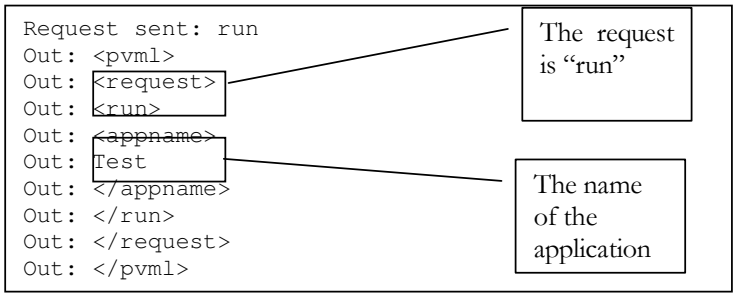

Figure 7-1 Engine sends run request

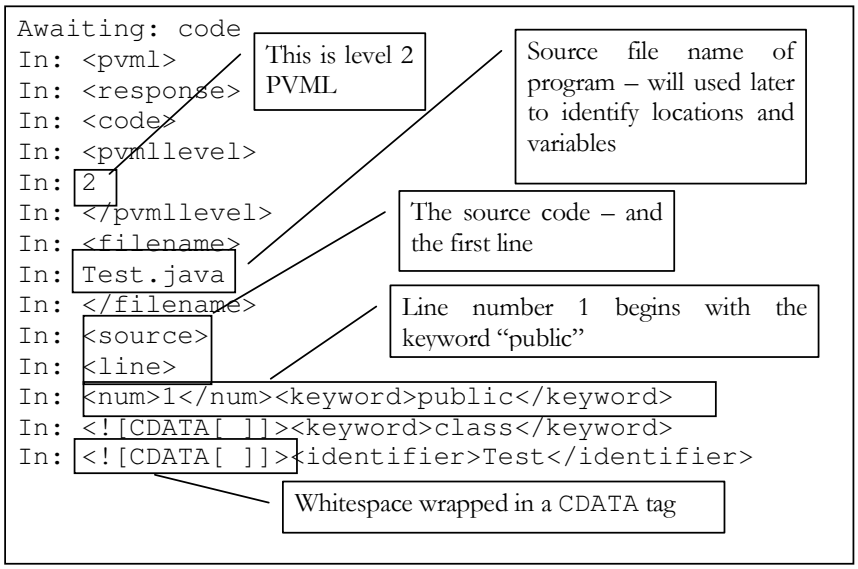

Figure 7-2 Start of code response

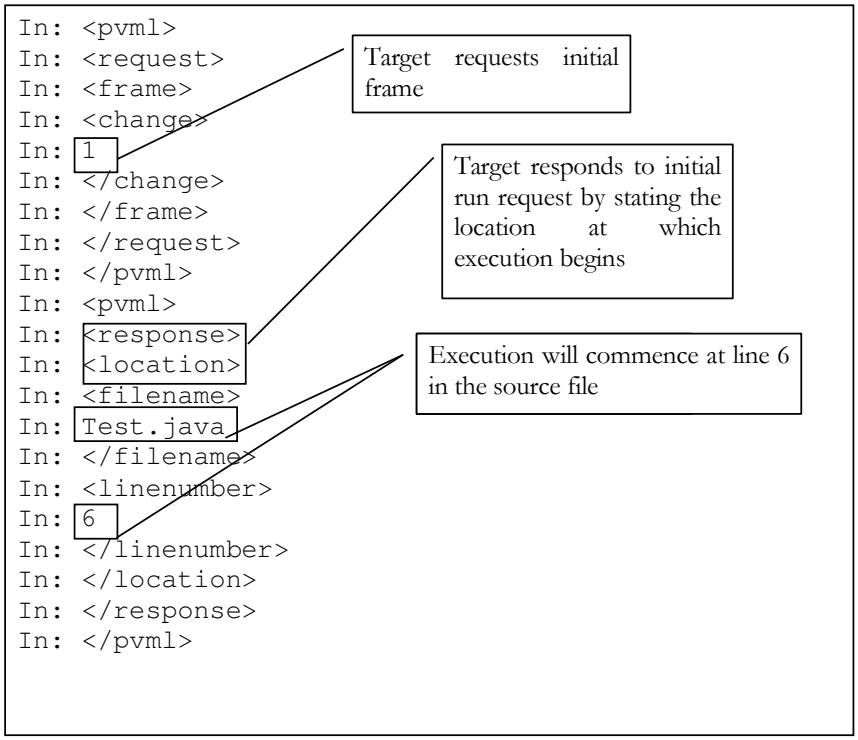

Figure 7-3 Establish the execution starting point

7.4.2 Loading a C Program

This example shows an engine requesting the loading of a C program at the target. The result will be the source code view shown in Figure 8-5 on page 133. In this case it can be seen that the requested `appname` is a target file system path

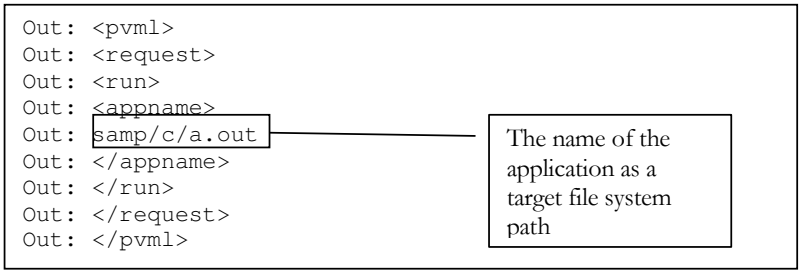

Figure 7-4 Engine run request references target file system

A sample line of C code, in this case a simple loop '`for(i=0;i<10;i++)`', as encoded in a PVML `code` response is shown in Figure 7-5.

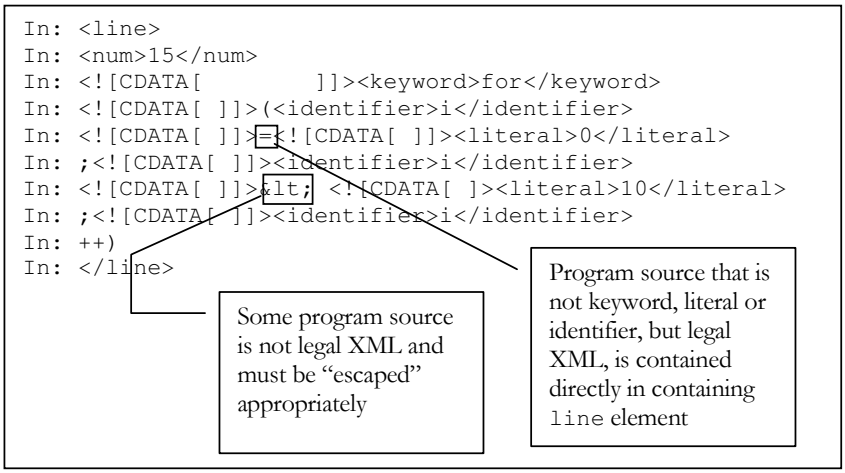

Figure 7-5 Sample line of C source code

7.4.3 Loading a FORTRAN program

The FORTRAN example shows a PVML level 1 `code` response. GDB identifies the source language of the target program as FORTRAN but no parser (see Section 8.3.1) is available to support a level 2 display. Without a parser available the entire source code is sent as an XML `CDATA` block. The result is the source code view in Figure 8-5 on page 134.

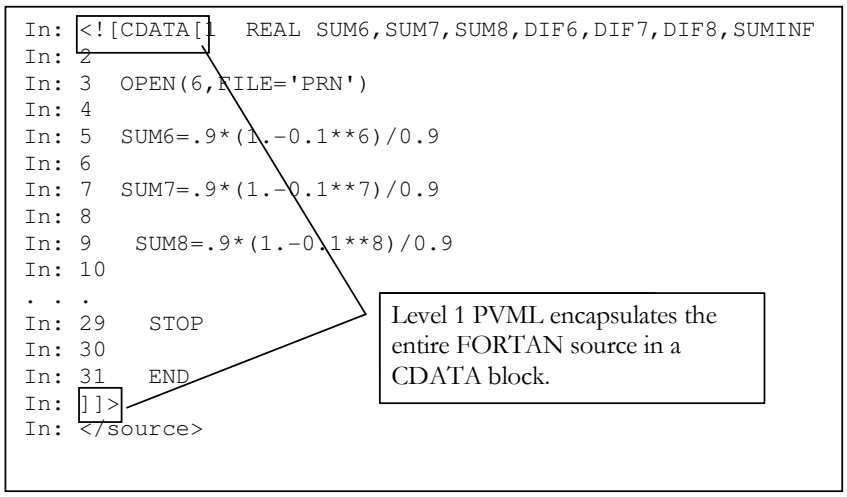

Figure 7-6 Level 1 PVML - FORTRAN source code

7.4.4 Single Step in a C Program

This example shows a single line of source code executing when a `step` request is sent to the target. The `numstep` parameter has not been set causing the default step size – a single line of source code.

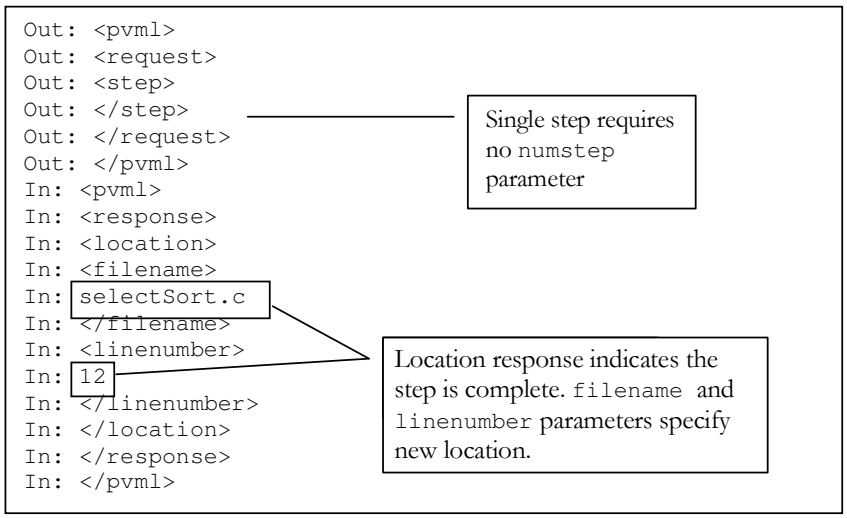

Figure 7-7 Single step in a C program

7.4.5 Single Step with a Frame Change

When stepping by a single source line causes a change of execution context at the target a `frame` request is sent. When a function in another source file has been called the subsequent `location` response will indicate a location in a source file which may not yet be cached at the engine. This will cause the engine to issue a `list` request to retrieve the new source code. The GUI view of this scenario is shown in Figure 8-4 on page 132.

```
In: <pvml>
In: <request>
In: <frame>
In: <change>
In: 1
In: </change>
In: </frame>
In: </request>
In: </pvml>
```

Figure 7-8 PVML frame request - adding an execution context

7.4.6 Placing a watch on a variable

The visualiser will provide a means whereby the user may select a variable to become part of the visual representation. Updates in the value of this variable need to be detected by the target. The implementation details of this process depend on the capabilities of the underlying debugger and the consequent design of the target PVML driver.

The `watch` request identifies a variable, using the language neutral terminology explained in Appendix A, and indicates whether a watch is being added to or removed from this variable.

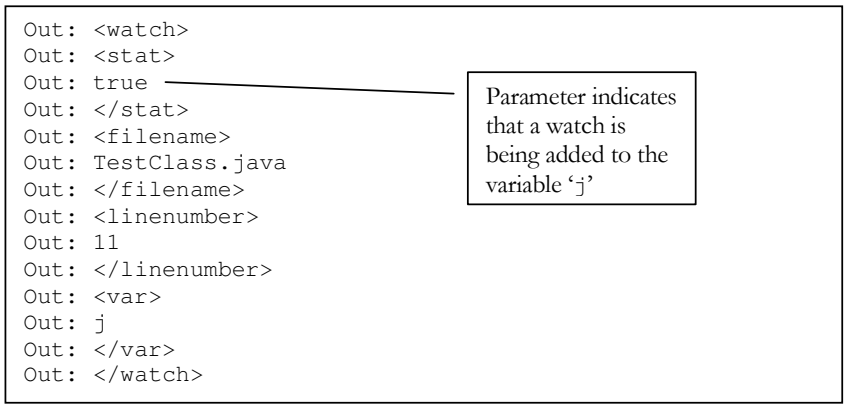

Figure 7-9 Adding a watch to a variable

7.4.7 Change in value of watched variable

The changed value of a watched variable will become available to the target asynchronously – as a result of program execution rather than action on the part of the user. The `data` request transmits the new value of the variable to the engine. PVML syntax, as explained in Appendix B, can represent arbitrary combinations of data values and data references.

Figure 7-10 shows the PVML that results from a simple variable (a Java `int`) update. The GUI view of this scenario is shown in Figure 8-9.

```
In: <data>
In: <filename>
In: TestClass.java
In: </filename>
In: <linenumber>
In: 11
In: </linenumber>
In: <varname>
In: i
In: </varname>
In: <value>
In: <type>
In: int
In: </type>
In: <val>
In: 1
In: </val>
In: </value>
In: </data>
```

Figure 7-10 A data request communicates a simple updated data value

Figure 7-11 shows the representation of a more complex variable. In this, more complex, case the formatting of the PVML in the figure has been manually altered to clarify the representation of the data values. The GUI view of this scenario can be seen in Figure 8-10.

```
<data>
  <filename>Test.java</filename>
  <linenumber>5</linenumber>
  <varname>var1</varname>
  <value>
    <type>TestClass</type>
    <value>
       <type>int</type>
       <val>51</val>
    </value>
    <value>
       <type>NestedClass</type>
       <value>
          <type>java.lang.Integer</type>
          <val>42</val>
       </value>
       <value>
          <type>java.lang.String</type>
          <val>Sample String</val>
       </value>
     </value>
  </value>
</data>
```

Figure 7-11 A data request communicates a complex variable update

7.4.8 Watched variable becoming out of scope

The visualiser has the responsibility of maintaining a visual representation, for the novice programmer, of variables that actually exist in the executing program. When a variable, that has been watched, is no longer in scope it is critical that the visualiser be made aware of this fact, in order that an appropriate visual reaction may ensue. Under these circumstances the optional `eoc` (end of context) element may be passed in place of a variable value in a `data` request.

```
In: <data>
In: <filename>
In: Test.java
In: </filename>
In: <proc>
In: morejunk30
In: </proc>
In: <var>
In: i
In: </var>
In: <value>
In: </value>
In: </data>
In: </request>
```

Figure 7-12 A watched variable becomes out of context

Chapter 8

REFERENCE ENGINE AND TARGETS

The foregoing PVML language scenarios have been based on actual PVML traffic between an engine and two different targets. This chapter describes the reference engine and targets. These reference implementations are all written in the Java programming language. The case has already been made in Chapter 2, for using Java to implement the engine – which ultimately will be the novice programming environment.

There is no fundamental reason, given the decoupling that PVML introduces, why the targets should be written in Java. In the context of this research, and the demonstration of a working PVML-based program development scenario, it has been prudent to take advantage of the fact that there are significant amounts of functionality that are shared between a target and an engine. The generation and parsing of PVML streams, along with the management of the network connections across which those streams flow, occur in the target and the engine and significant economies of effort have been achieved by using Java throughout.

In order to evaluate the use of PVML with a cross-section of programming languages, two targets have been created. A PVML target can most easily be characterised as one that encapsulates the functionality of a particular debugger:

- GDB target

  The debugger GDB has been mentioned at various points in the definition of PVML requirements. This debugger is almost universally available within UNIX systems and will debug programs written in a wide range of GNU supported languages. GDB provides a low-level, command line interface to symbolic debugging primitives for the languages that it supports.

  Many researchers have sought to develop enhancements to GDB functionality, both in terms of improving the user interface [29] and developing debugging languages [18],[78] but these efforts have not involved modification of GDB itself – rather the management of the command and output streams of the underlying debugger. The approach could be characterised as the development of *wrappers* for GDB and its functionality.

  In the context of this research the existence of a wrapper, Insight [85], that is written in Java, and which has open source, has been critical given the arguments already raised concerning Java.

- JDB Target

  The JDB debugger is part of the standard Java Development Kit (JDK) distribution. The debugger has a command line interface that is strongly modelled on GDB but the JDK also provides an Application Program Interface (API) to the full range of Java debugging functionality.

The reference engine does not set out to provide any program visualisation features. Instead the approach has been to provide a platform through which PVML debugging scenarios can be explored.

8.1 Shared Target and Engine Functionality

This section offers a more detailed examination of the extent of functionality that is shared by the reference engine and targets.

8.1.1 Generating PVML

The set of PVML requests and responses, that have been already discussed, are programmatically available to target and engine through a single module. PVML output is returned to the calling application as a Java String and the parameters, when required are passed as appropriate Java parameters.

8.1.2 Parsing PVML

The parsing of an incoming PVML stream relies, in the first instance, on libraries within the JDK that process XML documents. There are two distinct approaches to the parsing of XML streams – the Simple API for XML (SAX) parser and the Document Object Model (DOM) parser. These are discussed, in general terms, in Appendix E and a case is made for a particular combination of the SAX and DOM approaches. The result of this combination is that the PVML parser, that manages PVML specific aspects of the data stream, has access to a structured DOM representation of the request or response that is guaranteed to be clear of any empty nodes that could complicate processing.

Having used this combination of SAX and DOM parsing the PVML parser exposes the DOM version of the input (PVML request or response) to a series of calls that are made by the command processing loop of the engine or target. Some examples will make this clear.

Request or Response?

The top level loop, that first analyses the incoming PVML stream, must decide if the latest input is a request or response. In Figure 8-1 the incoming PVML is parsed into a DOM represented by the variable `doc`. The parser utility routine `getType()` will extract the value of the top-level element from the DOM indicating whether the input is a request or response.

```
Document doc = parser.parse( inString );
//Was this a request or response?
String inputType = parser.getType( doc );
if( inputType.equals( PVMLParser.REQUEST_TAG )){
```

Figure 8-1 Check whether input is request or response

Execute a Request

Having identified a request, the body of the request must be executed by the command interpreter at the target or engine. In Figure 8-2 the `CommandExecutor` class will cause this execution to occur in a separate `Thread` and to occur on the particular `interpreter`. The parser utility routine, `getNodeValue()`, will extract the body of the command from the DOM so that is may be passed to the command.

```
new Thread( new CommandExecutor( interpreter,
       requestType,
       parser.getNodeValue(doc,
             PVMLParser.REQUEST_TAG).
       getFirstChild())).
       start();
```

Figure 8-2 Executing a request

8.1.3 Socket Server

The underlying communication, at a transport level, is Transmission Control Protocol (TCP) traffic between Java implemented `Sockets` at the target and engine. The code that manages these `Socket` connections, establishing a connection and proceeding to process requests and responses is shared between the implementations of target and engine. This effect of this can be seen in Figure 8-2 where the `interpreter` variable, which represents the command processor that will handle the request, is a parameter in otherwise generic code.

8.2 The Reference Engine

This section describes the reference PVML engine in greater detail. The engine is based on the GUI debugger sample program that is part of the standard Java Platform Debugging Architecture (JPDA) [109] distribution. Sun Microsystems provides this sample, which implements a graphical interface to the underlying Java debugging API, in order to demonstrate the use of the API to debug a local Java program. The JPDA also implements, Java specific, remote debugging connections and these can also be used in the sample program.

In terms of this research the sample program, when separated at a layer that purely sends debugging requests and displays the replies, has provided a useful starting point for a PVML reference engine. The GUI has had features removed and added but is still recognisably that of the Sun sample program.

It must be stressed that the reference engine provides no data visualisation capabilities. Textual representations of watched data values are displayed. The addition of an interface between these values and an established, command driven, visualisation scheme such as JSamba [102] would provide such an ability, but this is beyond the scope of this research.

The reference implementation of PVML, described in Chapter 7, has been used to provide communication between the reference engine and targets. A PVML-based infrastructure that supports two important aspects of program visualisation has been demonstrated:

- Program source code

  The source language independent pretty-printing of program source code and the associated management of the display of the current execution point.

- Program data

  Any local or global program variable may be selected to be watched. Updates in variable values are displayed as human readable text.

8.2.1 Program Source Code

Several examples of the display of program source code and current execution point in the reference engine are shown here. These GUI examples correspond to the PVML scenarios that were introduced in Chapter 7.

Display of Java source code

This corresponds to the PVML in Section 7.4.1 and Figure 8-3 shows how Java source code is pretty printed. The pretty printing of source code is actually available for any language that can be parsed by the target.

A top-level pane is provided for each target connection made from the engine and within this pane a separate source code pane is provided for each execution context. The top-level frame is labelled ('`jdb@!localhost:12345`' in this example) with the name of the debugger in the current target and TCP/IP (host and socket) location information. The initial pane is numbered '1' – the first execution frame. The pane is also labelled with the name of the source code file.

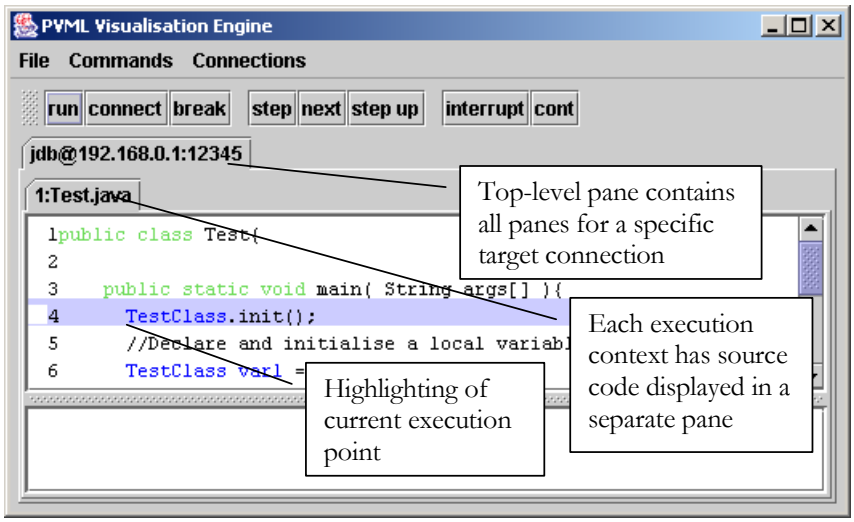

Figure 8-3 Engine displaying sample Java source code

As target program execution proceeds, and methods are invoked or functions called, new execution contexts will be entered. Each new execution context, as signalled by a PVML frame request (Section 7.4.5), will cause the engine to display a new source pane. If the function is defined in a source file that has not previously been displayed by the engine, this could result in the transfer of a new batch of source code.

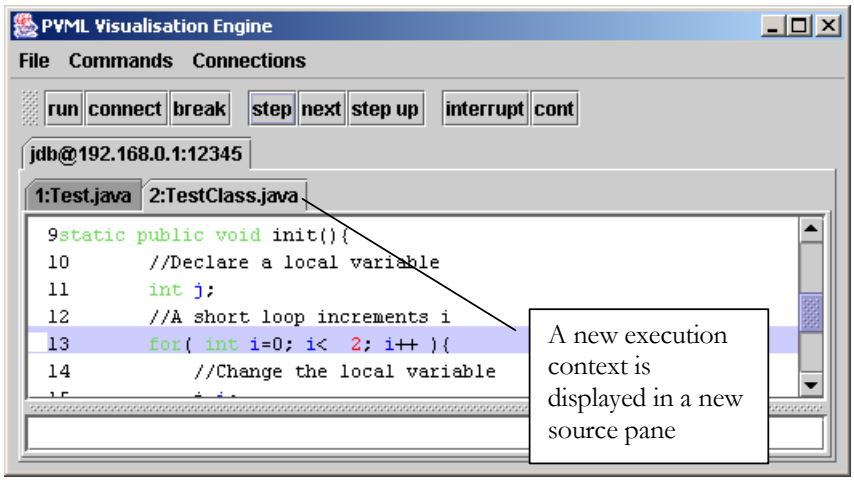

Figure 8-4 Engine displaying Java source code in a second execution context

Display of C source code

Figure 8-5 corresponds to the PVML in Section 7.4.2 and shows pretty printed C source code, that has been provided by a GDB based target that has access to a C language parser.

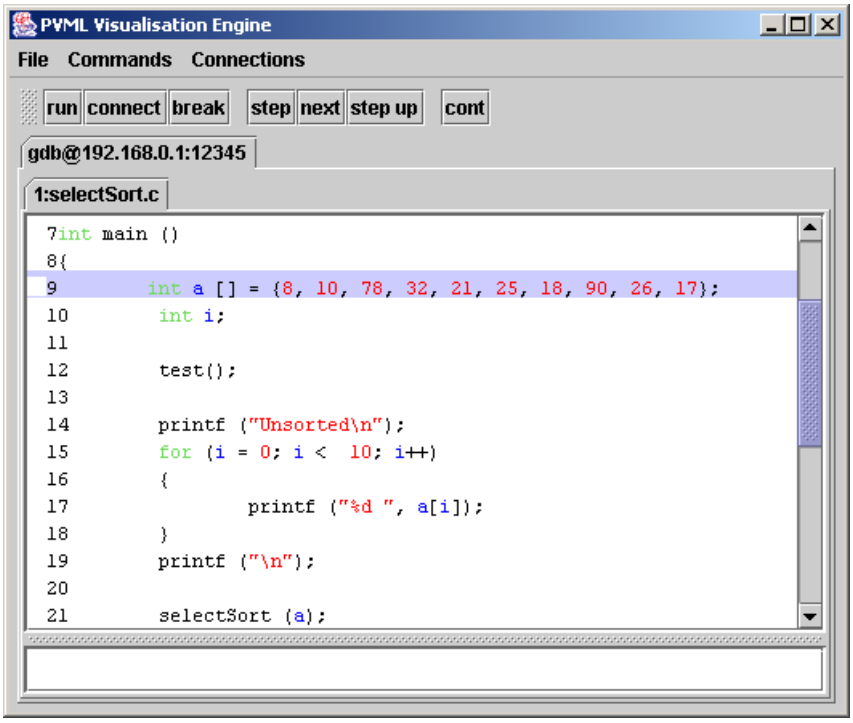

Figure 8-5 Engine showing sample C source code

Display of FORTRAN Source Code

If the debugger in the target, for some reason, cannot parse the source file the level of PVML used defaults to one which does not support pretty printing. This has been described in Section 7.4.3, where the PVML implications are shown, and will also be discussed from a target point of view in Section 8.4.2.

The reference engine allows the user to step through such a program but does not provide a means to select program variables to be watched. In the absence

of a program parse tree no automatic detection of program variables can be provided and the selection of a variable to be watched would need to be based on textual entry of a variable name.

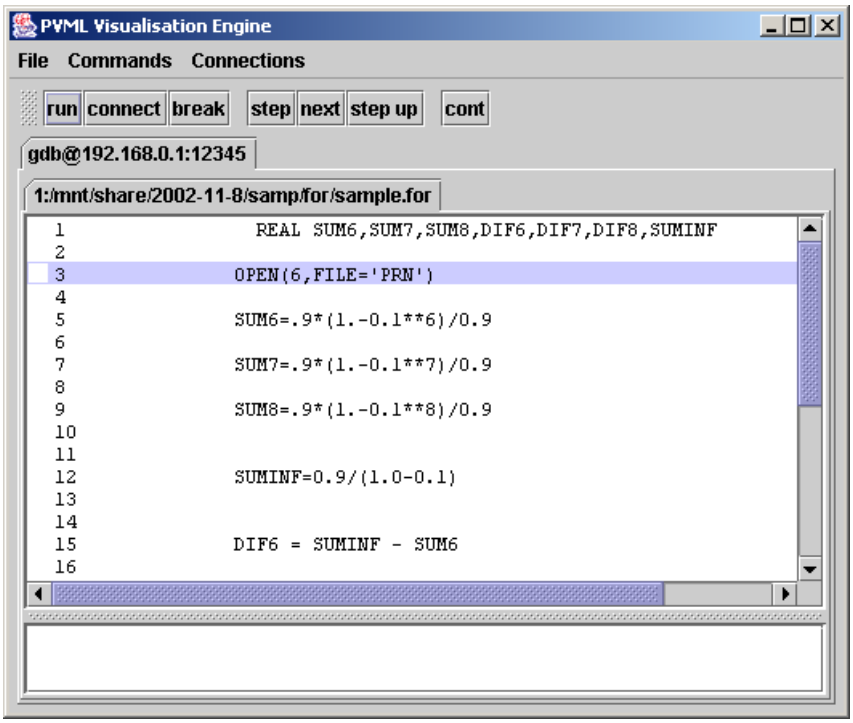

Figure 8-6 Engine showing sample FORTRAN source code

Simultaneous debugging in several different languages

The engine can connect to an indefinite number of targets, each of which may be directed to run a program written in a distinct source language. Figure 8-7 shows the reference engine being used with three targets – a JDB target running a Java program and two separate GDB targets, one running a program written in C and the other running a program written in FORTRAN.

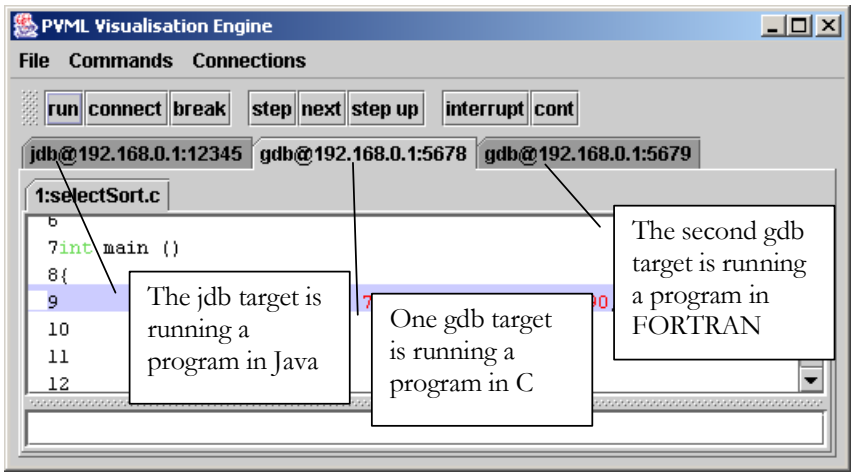

Figure 8-7 Simultaneous sessions in three source languages

8.2.2 Program Data

The reference engine allows any program variable to be selected to be watched by the target. The target implementation of variable watching is discussed in detail in Sections 8.3 and 8.5 where the specific reference targets are described.

From the point of view of the engine there are two issues that are addressed – the selection of a variable to be watched and the display of value updates.

Selection of a variable

When the target is able to parse the source language of the program, the pretty printing of the engine listing enables the engine to identify the declarations of variables in the program listing. A mouse click on a variable declaration will tag that program variable to be watched and the source listing is modified with all occurrences of that variable being marked with a border as in Figure 8-8. A mouse click on a watched variable will remove the watch. The corresponding PVML is shown in Figure 7-9.

The declaration of the variable is also highlighted which means that the attention of the novice programmer is drawn to the scoping rules of the

language. As can be seen in the associated PVML it is the line number of the declaration that is passed to the target in order to unambiguously identify the variable.

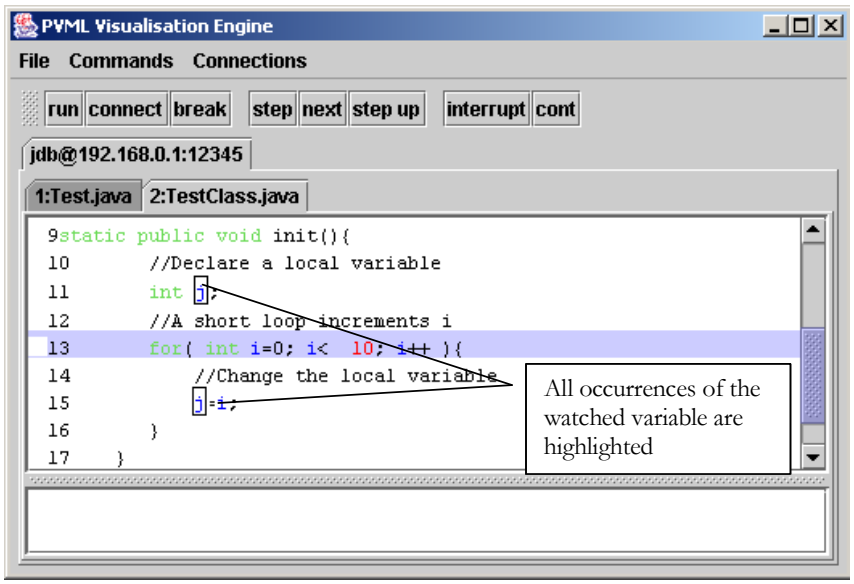

Figure 8-8 Engine showing a watched variable

Clearly this selection technique depends upon the pretty-printing of the source code which in turn depends on program parsing at the target. The reference engine does not provide a user interface to support the specification of variables to be watched in situations such as Figure 8-6 where the source program has not been parsed.

Display of variable updates

Variable updates, received in a PVML `data` request, are displayed by the engine in a raw, textual form, in the lower pane associated with each source code frame. In a visualisation context it is this output that would be parsed by a visualisation tool.

If a declarative approach to visualisation specification (Section 4.4) were adopted, particular elements of this stream would form the input to

expressions, the evaluation of which would result in specific visual consequences.

The, more widely adopted, imperative approach to specifying program visualisation would map updates in variable values directly to visual representations of those variables.

Figure 8-9 shows a simple variable value being displayed when program execution results in two new values being assigned to a watched variable. The PVML that results in this display is shown in Section 7.4.7.

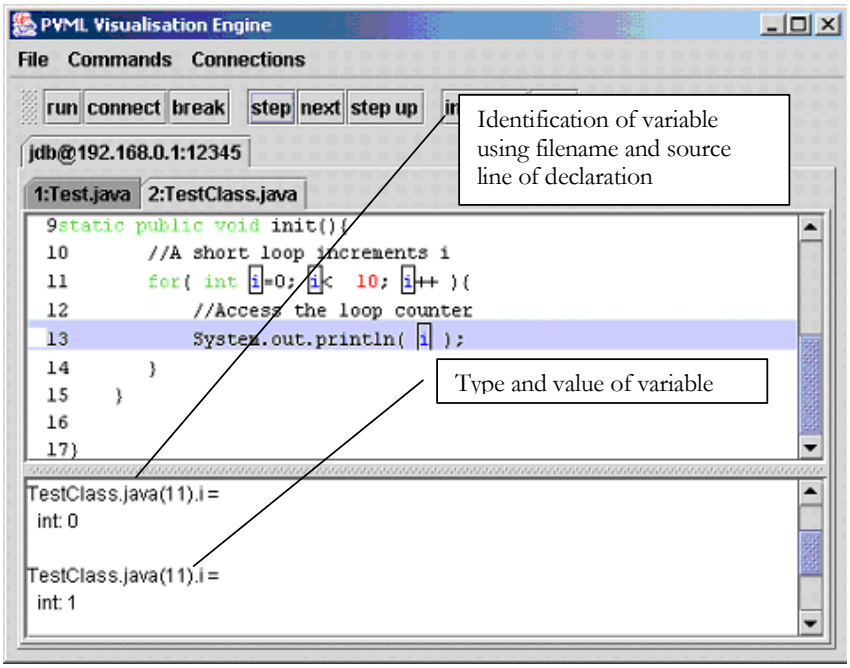

Figure 8-9 Display of a simple Java variable value

Figure 8-10 shows the output when the value of a more complex variable is watched. In this case the variable consisted of an instance of `NestedClass` within an instance of `TestClass`. The PVML for this transaction is shown in Figure 7-11.

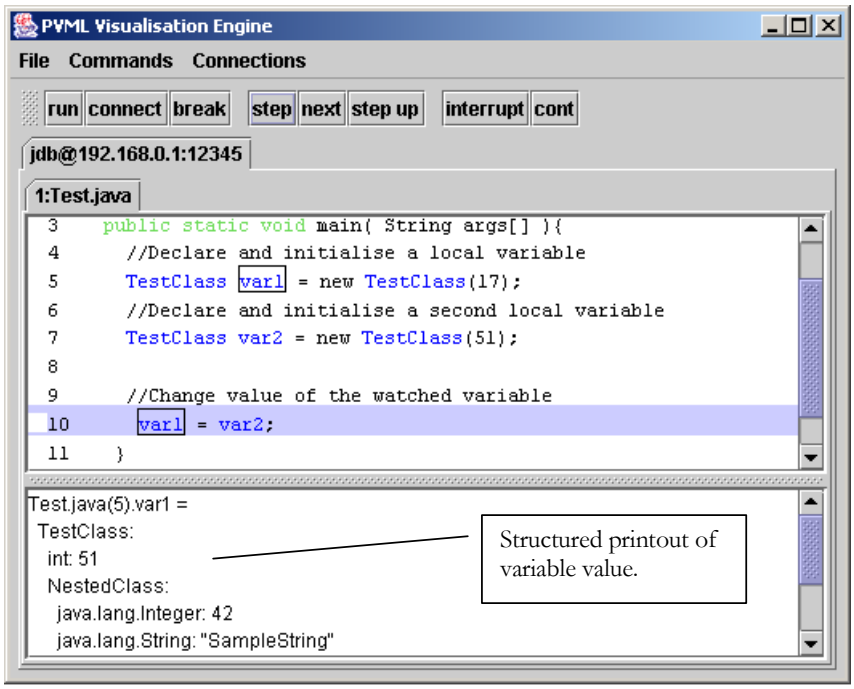

Figure 8-10 Display of a complex Java data item

A visualisation scenario would need to include a means to handle variables that have become out of scope – their display would need to be modified in some way (possibly 'greyed out') or else they might simply disappear. In the reference engine, since the effect of returning from a function call is to close the source window for that function, the values of variables within that function will also disappear from view.

Figure 8-11 illustrates the display that occurs when a variable is not longer in scope. This scenario actually exposes an interesting aspect of the scoping rules in Java. In Java it is normal to declare a loop counter in the manner shown in Line 13 of the sample program – with the expected consequence that, by Line 18, the variable '`i`' will no longer be in scope. This, however, is not the case as can be seen by the program execution highlight needing to be at Line 19 before the "Out of context" message is displayed. Java keeps a variable in scope for a short, unspecified, period after the block in which it is declared.

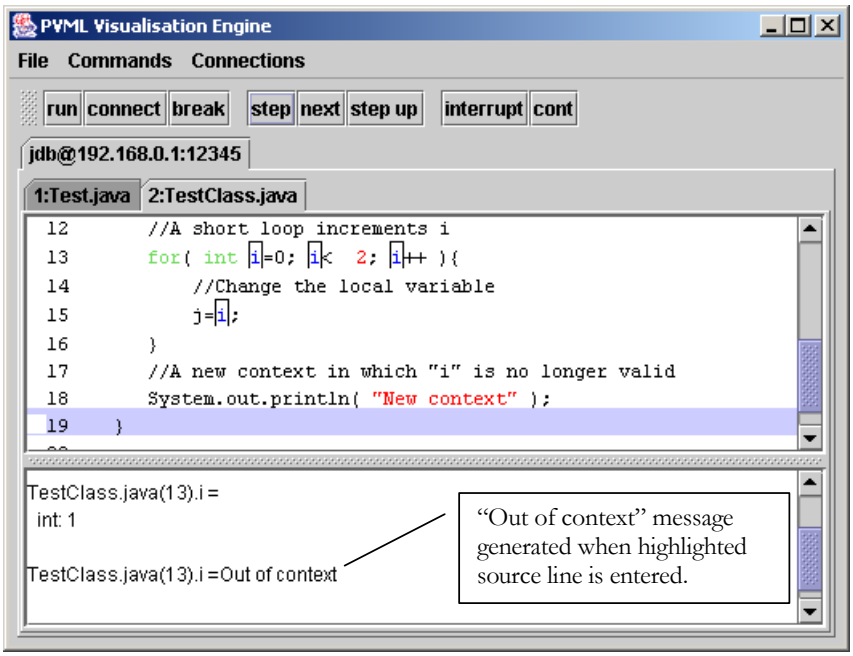

Figure 8-11 Display of a Java variable becoming out of scope

8.3 Common Target Components

It is fundamental to the PVML-based architecture that most target functionality is implemented in a PVML target driver that is specifically matched to a particular underlying debugger. It is the target driver that maps the commands of the abstract 'PVML debugger' to the command set of the particular debugger that is being encapsulated.

Two aspects of the target functionality are generic to all targets:

- Program Parsing

  The parsing of target program code, in order to generate the pretty printed source display, is generic to all targets.

- Watchpoint Management

  The variable watching functionality of the PVML debugger is a significant extension of that available in typical debuggers. Various classes that are shared by all targets manage this aspect.

8.3.1 Program Parsing

The necessary language parsers are written automatically by a 'compiler compiler'. JavaCC [121] is a Java implementation of a parser generator, that takes a language grammar representation as input and automatically generates the Java classes required to implement a parser for source files that adhere to the grammar. University Collage of Los Angeles maintains a repository of grammar files [111] for a cross-section of programming languages – a JavaCC grammar is defined in a file with a '`jj`' extension containing productions that are very similar to those expressed in the Backus-Naur Form (BNF) definition of a language syntax.

Generation of the PVML representation of a program source file requires that the source file be parsed into a tree representation which is traversed in such an order that a correct XML representation of the source code is output. The tree representation can be generated automatically from the JavaCC grammar definition using the Java Tree Builder (JTB) [52] which extends the parsing functionality of JavaCC to include the building of a parse tree. JTB also provides methods, that make use of the Visitor pattern [30], to enable classes to be written that will perform certain actions at nodes of the parse tree.

In this context an `XMLTreeDumper` has been written which generates appropriate PVML to describe each region of the parse tree. A distinct `XMLTreeDumper` must be provided for each source language that is supported since this class explicitly references the productions of the source language.

Figure 8-12 shows fragments extracted from the `XMLTreeDumper`'s written for Java and for C. In both cases the Visitor method displayed is the one that

is called at the root of the parse tree. In the case of Java this is represented by a `CompilationUnit` node whereas in C the representation is in the form of `TranslationUnit`. At this level in the tree the functionality required is identical – namely to recursively visit the rest of the tree before closing any open XML elements. Visitor methods for nodes lower down in the tree may differ significantly according to source language.

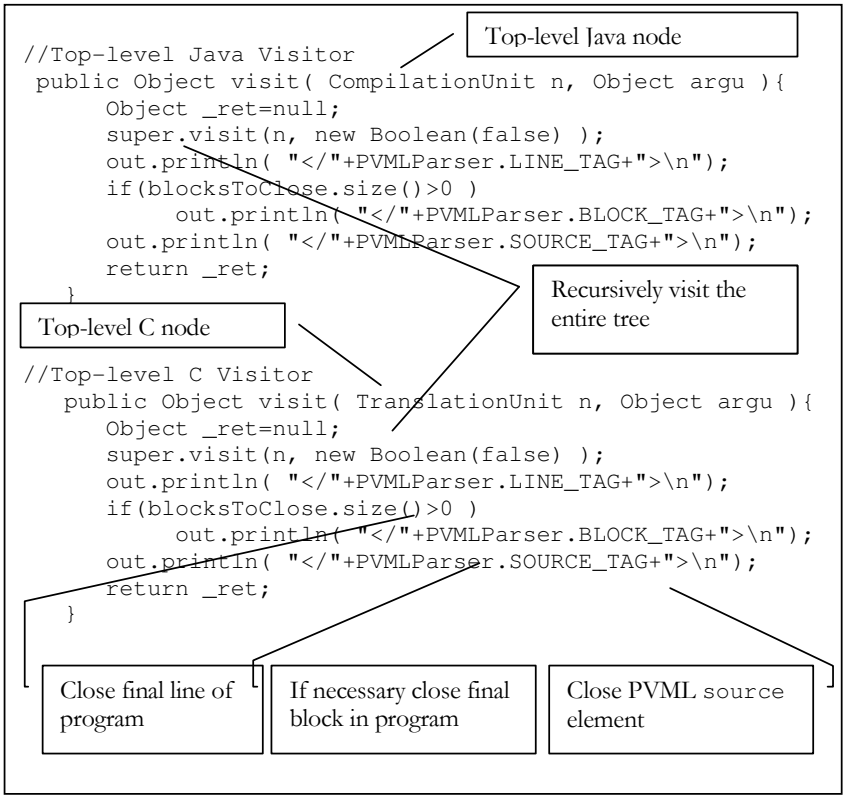

Figure 8-12 `XMLTreeDumper` fragments for top-level node Visitor in two source languages

8.3.2 Parser Modifications

In most respects the automatically generated parser and the associated, custom-written, Visitor class combine to produce the necessary PVML output. There are two considerations though which lead to modifications to the JTB-written parser code:

Program Comments

It is fundamental to the operation of a parser that program comments are ignored and do not appear in the parse tree of the source code. However, from the point of view of the novice programmer, it is important that the comments are displayed in the engine. The `ParserTokenManager` class, written by JTB, is modified such that, when comment tokens are encountered, their text and position in the source code are logged as shown in Figure 8-13 with a `CommentManager` class.

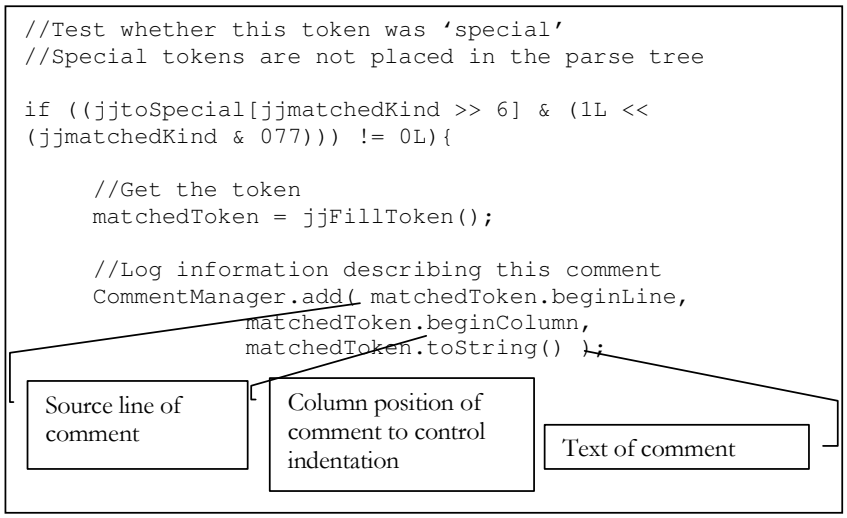

Figure 8-13 `ParserTokenManager` saves source code comment information

The `XMLTreeDumper` checks with the `CommentManager` class before generating the PVML for a new source line and any, outstanding, comments are returned and inserted in the PVML stream as `comment` elements which will be rendered appropriately by the engine. This is shown in Figure 8-14.

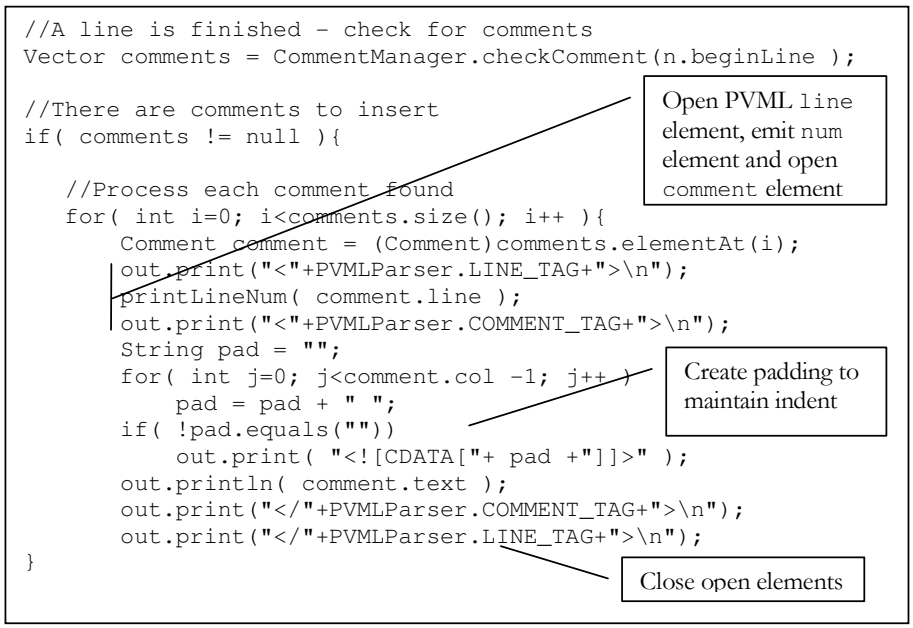

Figure 8-14 `XMLTreeDumper` reinserts source comments in PVML stream

Parsing Multiple Source Languages

A target, such as the GDB target described below, needs to be able to parse more than one source language and hence have access to JDB-generated parsers for a number of languages. The selection, amongst these parsers, will depend on the debugger reporting the source language of the current debug target.

To implement this feature the JTB invocation that generates the parser classes is directed, through a command line switch, to create these classes in a Java package, the name of which includes the name of the source language as shown in Figure 8-15.

The target command interpreter is able to construct the name of the required parser and tree-dumper classes and attempt to load these classes at run-time. Failure to locate a parser for a program source language will cause the target to fall back to a PVML Level 1 representation.

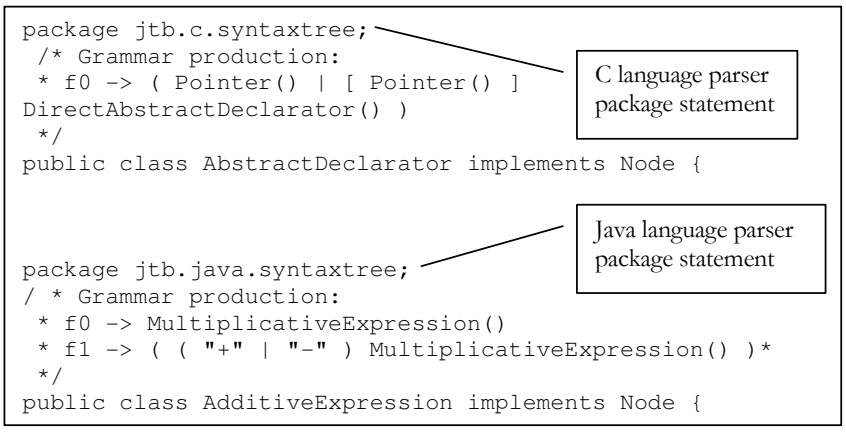

Figure 8-15 JTB-written code fragments showing language dependent package

8.3.3 Program Watchpoint Management

PVML sets out to provide debugging functionality that is independent of source language issues and which is also, from the point of view of a novice programmer, uniform in application across all aspects of their target program.

This aspect of PVML is most evident in terms of the watching of program variable updates. Table 8-1 sets out the contrasting approaches to variable watching in the PVML debugger and the two target debuggers that have been evaluated and it is the task of the `WatchManager` class to map the generous and uniform PVML watch model to the heterogeneous watch models of the supported debuggers.

To provide program variable watch support at points in program execution where the debugger would not (local method variables in JDB, out-of-context variables in GDB) requires the `WatchManager` to maintain data structures that record details of watched variables independently of the underlying debugger.

Debugger	Variable watching functionality
PVML	- Any program variable can be selected to be watched - Variable is specified by source file and line number of declaration - Variable updates and 'out of context' messages returned to engine in `data` requests - Visual treatment of `data` requests is delegated to the visualiser role
GDB	- Any, in context, variable can be selected to be watched - Variable is specified by name - Automatically deletes watchpoints for variables that become out of context
JDB	- Class 'members' can be watched (variables that are defined at the head of the class) - Local method variables cannot be watched - Variable to be watched is specified by name

Table 8-1 Contrasting debugger approaches to program variable watching

The `WatchManager` maintains data structures relating to potential program watch points and through access to these structures the target driver is able to command the debugger appropriately to watch variables that would otherwise be unavailable. The following two examples clarify this process:

GDB – persistent watch on local variable

Since GDB can only be commanded to watch an in-scope variable the persistent watching of a variable that enters and leaves scope requires a new watch command upon each entry to that context. The target driver, upon entering a new context, checks with the `WatchManager` for any variable watches that need to be re-established.

JDB – watching local variables

JDB cannot set a watch on the local variables of a method. In order to watch such a variable the target driver needs to command JDB to step by machine instructions in regions where a watched variable exists and manually inspect the value of that variable. This is a significant performance overhead and

should be avoided in the absence of any variables that need to be watched. The WatchManager is designed to avoid un-necessary low-level stepping.

8.3.4 The `WatchManager`

The WatchManager maintains data structures that describe each scope in the target program in order that information regarding watched variables can be stored independently of the target debugger and in a manner that optimises the target driver commands that are sent to the debugger.

The description of a particular program scope is maintained by an instance of the `ProcBlock` class. A `ProcBlock` is given a name according to the rules described in Appendix A. The `WatchManager` maintains a `Hashtable` of `ProcBlocks` for each source file in the target program that is indexed by `ProcBlock` name, and from which the status of watched variables may be retrieved by the target driver upon entry into a context.

The `ProcBlock` class

`A ProcBlock` instance stores details of all variables in a context that are watched. The members of this class store the information needed to manage the life cycle of variable watching for a particular scope in the source program:

- `watchcount`

  This integer stores the current number of variables in a scope being watched. On entering a scope the target driver check this count and if variables are currently being watched proceeds to command the debugger using the less efficient, low-level command set.

- `filename, procname`

  The combination of the source filename and the PVML scope name generated according to the techniques described in Appendix A, uniquely identify this scope within the target program.

- `vars`

  This `Hashtable`, indexed by variable name, stores details of watched variables in this scope.

- `startLine, endLine`

  The source code lines included within this scope.

8.4 The GDB Target

This section describes the PVML GDB target in greater detail paying particular attention to the use of Insight [85], the open source, Java wrapper for GDB.

Many research projects and developments have involved extending the behaviour of GDB as noted in the introduction to this chapter and also in the discussion of debuggers in Chapter 5. These endeavours have largely been based on choosing not to modify, or directly invoke, GDB functionality but instead, to feed commands to an underlying GDB invocation and to capture the resulting GDB output. Such an approach is sometimes referred to as 'screen scraping'.

Insight is a Java GUI front-end for GDB which incorporates this screen scraping approach. In the context of a PVML target, the GUI is dispensed with and the low-level Insight classes, which control access to GDB, are built into the target driver.

Insight extends the event-driven architecture that typifies Java GUI's by implementing a `Panel` class. This class responds to asynchronous events

generated by an underlying `GDBServer` class that directly manages GDB input and output. Insight defines many sub classes of `Panel` that register with `GDBServer` and subscribe to certain classes of event. Callbacks from the server then result in appropriate GUI updates taking place. In the context of creating a PVML target driver for GDB, this architecture is particularly well suited to the extension that has been implemented.

As has been previously described on page 129, the parsing of the incoming PVML stream is handled by a `PVMLParser` class, that is common to both engine and target implementations. The `PVMLParser` passes commands onwards to an instance of a `CommandInterpreter`. The GDB target includes a `pvml.target.gdb.TargetCommandInterpreter` class that implements the commands received. This class maintains communication with the active GDB invocation through sub-classes of the Insight `Panel` class that register with the Insight `GDBServer` and receive responses from GDB.

Additional communications, that are not supported by the Insight infrastructure, take place through the direct invocation of methods of the Insight `GDBServer` object.

The aspect of target design that requires detailed discussion here is the mapping between the command set of the abstract PVML debugger and that of the underlying debugger, in this instance GDB. This relationship is described in Section 8.4.1.

Other issues that are particular to the use of GDB in a PVML target are discussed in Section

8.4.1 PVML to GDB Command Mapping

PVML debugger request	GDB debugger command	Comment
`break`	`break`	The only type of breakpoint specification in PVML is by filename and line number.
`cont`	`cont`	Directly mapped
`data`	–	This request returns data values to the engine. All watches, when the variable is in context, are native GDB watches. The `WatchPanel` class receives notification of the update of watched variables and forwards a PVML `data` request to the engine.
`frame`	`backtrace`	The PVML frame request is an asynchronous indication of a frame change for which there is no equivalent GDB response. A frame change in GDB is detected by an invocation of the `backtrace` command following each `step` command. When a frame change is detected Insight notifies an instance of the `FramePanel` class which forwards a PVML `frame` request to the engine.
`list`	–	Program listing in a PVML target is not implemented through the debugger. The target driver directly reads the source file – through a language parser if one is available.
`next`	`next`	Directly mapped
`read`	`print`	The PVML `read` command can only be applied to variables whereas GDB can evaluate an expression in a supported source language.
`run`	`run`	Directly mapped – before running a program a breakpoint must be set at the entry point to the program.
`step`	`step`	Directly mapped – a `backtrace` command is included to detect frame changes.
`watch`	`watch`	The GDB watch command allows a watch to be set (or cleared) on any, in context, variable. The PVML watch command allows a watch variable to specified by source filename and line number – in other words regardless of context. The PVML watch is set in the `WatchManager`. If the variable is in context the GDB watch is set as well – otherwise the setting of this watch in GDB is delegated to the `WatchManager`.
`write`		The PVML `write` command can only be applied to variables whereas GDB can evaluate an expression in a supported language and assign the result to a variable...

Table 8-2 Mapping PVML debugger requests to GDB

8.4.2 GDB Target Issues

Some aspects of GDB, and the GNU language environment, have a particular impact on the design of the target driver and of PVML.

Source Language Identification

The GDB '`info source`' command returns the name of the program source language of the currently executing source file. The target driver uses this command to retrieve the name that it uses to construct the language parser class name as described on page 143. If the parser class cannot be found the target will default to a PVML Level 1 representation and there will be no pretty printing of the source code available at the engine.

GDB source language identification depends upon the extension used in the source filename (the part of the filename after the last period in the name) and situations where source files have been given non-standard extensions will prevent PVML Level 2 from being used, even if, in fact a parser exists for the source language.

Program Entry Point

If single stepping, rather than full speed execution, is required a break point must be set at some point in the target program before it is run – the default behaviour of GDB is to run a target program to completion. It is normal to set a breakpoint at the first instruction of a program before running it under GDB in order that initial control is passed to the debugger.

In the case of a PVML target this breakpoint needs to be automatically set in order that programs respond to the PVML `run` request by loading and advancing to the first line of user source code.

The automatic, language-independent setting of this initial breakpoint is complicated by the fact that different source languages may use a different symbol name to identify the entry point to the initial source file of the

program. The executable file for a program written in a language such as GNU FORTRAN, which is actually implemented in C, does have the normal C `main()` entry point, but the execution at this stage is within the libraries that support the FORTRAN environment. For the novice programmer the perception needs to be of execution commencing in the FORTRAN source code. This latter entry point, for FORTRAN, is named `MAIN__`.

This issue is resolved by attempting to set the initial breakpoint at all of the known program entry points as the fragment of code in Figure 8-16 shows.

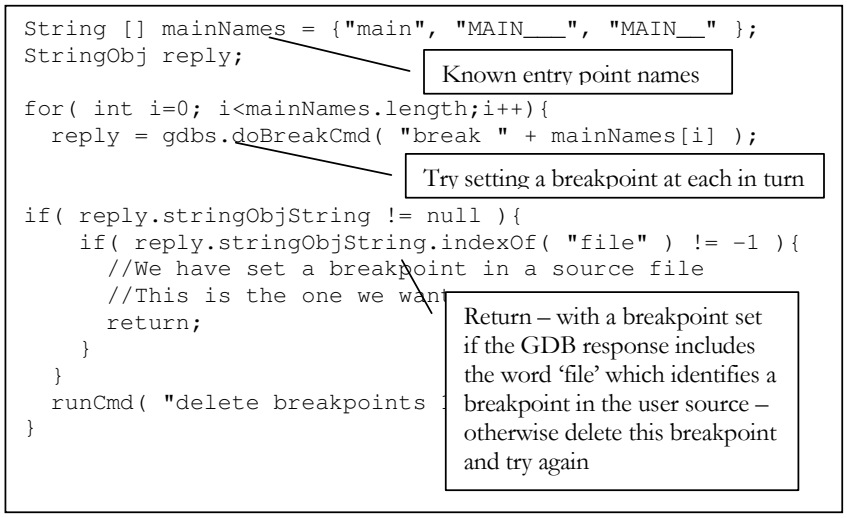

Figure 8-16 Setting an initial breakpoint

8.5 The JDB Target

This section describes the PVML JDB target in greater detail and pays particular attention to the relationship of this work the Java Platform Debugging Architecture (JPDA) [109].

The JPDA exposes all aspects of an executing Java program to programmatic manipulation. JDB, the Java debugger, was originally a stand-alone application that provided a command set that was very similar to that of GDB – but restricted to Java target programs. The publication of the JPDA,

which includes the Java Debug Interface (JDI), allows all the functionality of JDB, for example, to be provided in a sample Java program that is part of the JDPA library. The classes and interfaces that make up the JDI provide access to all the functionality that is needed in a PVML target that specifically hosts Java programs.

The limitations of JDB in relation to watching local method variables, as detailed in Table 8-1, can be seen as being related to the set of events defined in the `com.sun.jdi.event` package [109] which includes a `ModificationWatchpointEvent` that is fired when a class field is modified but no event that corresponds to modification of a local method variable.

8.5.1 PVML to JDB Command Mapping

PVML debugger request	JDB debugger command	Comment
`break`	`break`	The only type of breakpoint specification in PVML is by filename and line number.
`cont`	`cont`	Directly mapped
`data`	–	This request returns data values to the engine. Field watches are native to JDB whereas local variable watches are implemented through the WatchManager as described in Section 8.3.3.
`frame`	`trace methods`	The JDB debugger will announce frame changes when configured to do so with the `trace methods` command. This command enables the `MethodEntryEvent` and `MethodExitEvent` of the JDI, which announce frame changes asynchronously.
`list`	–	Program listing in a PVML target is not implemented through the debugger. The target driver directly reads the source file – through a language parser if one is available.
`next`	`next`	Directly mapped
`read`	`print`	The PVML `read` command can only be applied to variables whereas the JDI can evaluate a Java expression.
`run`	`run`	Directly mapped – before running a program a breakpoint must be set at the entry point to the program which can be done though the JDI.
`step`	`step`	Directly mapped
`watch`	`watch`	The JDI only allows class fields to be watched. Watching of other variables is implemented through the WatchManager as described in Section 8.3.3.
`write`	`set`	The PVML `write` command can only be applied to variables whereas the JDI can evaluate a Java expression and assign the result to a variable...

Table 8-3 Mapping PVML debugger requests to JDB

Chapter 9

DISCUSSION & FUTURE WORK

This chapter discusses the significance, limitations and possible future development of the research described in this thesis. The significance of PVML as a concept, and of the reference implementations that are included in this research, is described in the context of existing work in the fields of PV and remote debugging. There are important aspects of PVML that have been set aside as being beyond the scope of this thesis and some consideration needs to be given to the validity of the limitations that have been placed on the scope of this work.

This thesis is a component in the research portfolio presented herewith, in fulfillment of the submission requirements of the professional doctorate degree. The significance of this thesis, within the broader context of the portfolio, is described in the commentary contained in the portfolio. Particular attention is paid there to the thematic linkage that exists between all the work undertaken in this degree. The research presented in this thesis constitutes the culmination of that thematically linked program of study and represents a little over half of the entire work undertaken in the degree. Accordingly it has been necessary to limit the scope of the research undertaken here and the ensuing discussion of limitations in the PVML approach will draw attention to these limits.

9.1 The Significance of PVML

As described in Chapter 3, program visualisation has, in general, been based on monolithic systems that offer the ability to visualise execution of a program in a specific source language. The user interface through which the novice programmer gains access to these features is particular to the visualisation system. The consequences of this architecture are twofold:

- Novice programmers, as they move on to learn subsequent programming languages, will need to become conversant with yet another programming and visualisation environment. The work of Hendrix [41], discussed in Section 2.3.3 draws particular attention to this issue.
- The activities of the visualiser role – namely the design of pedagogically effective visual representations of program execution – are most usually undertaken by the developer of the PV system. There is no clear location for effecting changes in visualisation strategies that is independent of PV system design. This question has been explored in Chapter 4.

Both of these consequences support the idea of decomposing PV systems into more strictly decoupled modules. Through such decoupling, a scenario can be realised, where each of the three key PV roles, visualiser, programmer and user, interact with a distinct module in the system. This is not a new way to approach PV. It has been strongly argued for by Roman [88] who, along with several other researchers, has implemented PV systems that are decoupled along these lines.

The PVML proposal has similar decoupling boundaries but is distinctive in suggesting that the communication at these boundaries be in a standard and open format. The design of an extensible language, that permits arbitrary visualisation targets and engines to interact, potentially allows many existing visualisation components to interoperate. Through enforcing a formalisation of functionality, program visualisation becomes open and extensible. This line of argument closely mirrors the developments in distributed computing, described in Section 7.1, which have seen the coupling between distributed components become less tight at a programming language level. PVML represents a significant addition to the expanding range [74] of XML-based initiatives that can implement this looser coupling between distributed components.

It is important to note that the boundary, across which this decoupling is proposed, is fundamentally one that is only traversed by program state information. Any intrusion of visual representation information into this flow would represent a division of attention for the visualiser, between the target and the engine. Roman has described the declarative model of visualisation in which "complete access to program state" provides the input that is required for declaring visualisation mappings from "programs to pictures". PVML has the express purpose of delivering such state information to the visualisation engine.

This reasoning raises the question of whether PVML should also be located within the domain of debugging languages. Precedents have been cited for building visualisation environments around debuggers (see the introduction to Chapter 5) and this proposal proceeds in that vein. In general, a debugging environment can deliver arbitrary amounts of program state information. In a PVML-based environment this same information will be available to an engine, and hence the visualiser, through typical, generic Internet connections.

A significant effort has been made in Chapter 5 to relate the design of PVML to the literature describing debugging languages. It is shown that, whilst there is a significant overlap in functional requirements, PVML introduces distinct considerations. The PV motivation, especially the emphasis on the needs of programming novices, constrains the breadth of coverage of the debugging domain. Furthermore, the truly decoupled nature of the target and the engine extends remote debugging beyond its normal boundaries.

The true significance of PVML will become apparent as engine and target drivers are developed for a variety of existing components. In some instances these developments will require extensions to the PVML language, where the appropriate interactions move beyond what has been considered generic amongst programming languages, into more language specific aspects.

Within this thesis the case of the 'object test bench' has been considered. This is an example of a language specific extension to PVML, in this case one that could be applied to object oriented languages.

9.2 Some Criticisms of PVML

The scope of this study has focused attention on a generic core of functionality for PVML and excluded certain important areas. This study also includes a reference implementation of the PVML language that is based on XML.

This section explores the rationale behind a number of exclusions and provides a brief discussion of the issues that would be involved were future development to be undertaken in such areas. The design decision to base PVML on XML is also discussed critically.

9.2.1 Novices and Experts

One aspect of this work that requires some mention is the decision that was made, at an early stage, to focus on the requirements of novice programmers. On the one hand, as has been shown in Section 3.7, the evaluation of the effectiveness of program visualisation has largely focused on its use by novices. An interest in visualisation is to a considerable extent, as far as the literature is concerned, an interest in programming by novices.

It has been argued in Section 2.3.2, that the feature-richness of the programming environment be deliberately curtailed when novice use is considered. By setting aside complex features, the design of PVML becomes a realistic undertaking within the scope of this research. Future work in this field can examine the application of these techniques to professional programming environments but this work would undoubtedly raise many new issues.

As described in Section 5.3, the generic core of PVML, which can be generalised across several paradigms of computing language, is a subset of complete debugger functionality. In this sense it is reasonable to consider PVML as providing an abstract debugger which implements a set of features that are appropriate to novice use. The exclusion of features that is implicit in the approach of Johnson [48] or Hanson [38] is emulated by the PVML proposal.

9.2.2 Granularity

Although the topic of granularity has been discussed in defining the general requirements for PVML, the reference language, engine and targets do not put any of these ideas into practice. A convincing demonstration of filtering the PVML stream was considered beyond the scope of this research but the theoretical functionality is present. Filtering should ensure that only changes in state are transmitted and that the level of detail in that state can be controlled by the engine. Filtering does not affect the fundamental concept of the abstract debugger but it does have the potential to substantially impact the usability of a PVML-based system. Reducing the data throughput of the PVML connection will increase the responsiveness of a PVML engine and lessen the impact of PVML-based visualisation on computing infrastructure.

Granularity is also an important aspect of PV, as discussed in Section 6.7, since the visualiser needs to manage the scale and scope of visual artifacts that are presented to the user. Control over the extent and detail of visualisations offered to a potentially struggling, novice programmer, is a significant pedagogical issue that [83] referred to as elision control. In future, PVML would need to be extended to include terms through which an engine, under the direction of a visualiser or perhaps as the result of user selection, transmits filter specifications to a target. The consequent limiting of PVML traffic to that which is strictly necessary to support a particular visual representation could clearly have an impact on system responsiveness.

9.2.3 Use of XML

The design of PVML, as a specialisation of XML, has a number of particular consequences. There are several aspects of the XML approach to data representation that represent potential criticisms of the PVML design presented and therefore require further discussion.

Well-formedness

Each PVML communication is a complete XML document, which is required to be 'well-formed', in the sense of having correct structure. A well-formed document is defined by the recursive application of two rules:

- Elements that are opened must be closed.
- A nested element must be closed before its parent element is closed.

The representation of program code in XML does not raise any problems – program code also adheres to the 'well formed' principle and hence the PVML encoding of program code can be justified.

Program state, during execution, can also usually considered to 'well formed', in the sense that context entry is matched by context exit according to similar rules that govern XML parsing. There are, however, circumstances in program execution that, at first sight, do not meet this requirement – such as an instruction to 'goto' a label, an exception or error condition or a pause for user input. The overall state of the executing program at such junctures is no longer one that can necessarily be described in a valid XML document.

It remains to consider whether this theoretical problem is, in fact, an actual problem in the use of PVML. The decoupling between an engine and target is such that the need never, in practice, arises to generate a complete PVML description of program state. An examination of the PVML command set shows that, even though the target program may, at times, be in an 'ill formed' state, PVML only communicates a fraction of that state – the current location – which can always be expressed in a well formed manner.

Representation of binary data

Computer programs fundamentally manipulate binary data. An integer value is represented by a number of bytes of memory; a string value by a series of bytes storing the encoding (formerly single byte ASCII but often now two-byte Unicode) of the characters in the string.

XML, a 'text-based' language, can only include 'printable' bytes in a valid document. In the case of the two examples given, integer and string values, the representation in an XML stream is straightforward. XML allows a character coding format to be declared in a document and hence any Unicode represented strings can be included in a document. Integers, and other simple data types, can be adequately represented as strings – at the cost of some verbosity relative to their binary forms.

A problem arises though, with more extensive binary data, such as images. As has been noted in Appendix B, the designers of XML consider this question to be beyond the scope of the XML standard. It is intended that the incorporation of binary data into an XML stream be based on established Internet standards such as Multi-purpose Internet Mail Extension (MIME).

As discussed in Section 2.3.4 the PVML-based model of PV explicitly targets use in a generic Internet context, where the target and engine may be on either side of an arbitrary extent of security-related firewall architectures. In this context the encoding of binary data according to extant standards would be a requirement, rather than an impediment.

Verbosity

Attention has already been drawn to the, often verbose, encoding of simple binary data, such as an integer, when sent in an XML stream[3]. The discussion of MIME encoding, in which eight bit bytes are encoded as seven bit bytes

[3] A two-byte integer could represent a five-digit number, which would occupy five bytes when encoded as an ASCII string or even ten bytes as a Unicode string.

would increase the size of data by between 10 and 20%. However the verbosity of XML extends beyond data encoding to the fact that XML includes 'meta data' in the stream. In the context of PVML the occurrence of, for example, the element name `<request>` at the beginning of a PVML utterance and the closing tag `</request>` represents a significant overhead.

A practical PVML-based PV system could employ various strategies to reduce the volume of this meta-data:

- Condensed DTD

  For the sake of clarity in this research, element names have been full, explanatory, words. A working system could be based on an alternative DTD with considerably abbreviated element names.

- Attribute Normal Form

  Again for the sake of clarity, all representations herein have been in what is known as the 'Element Normal Form' (ENF) of XML. This form, in which elements simply contain other elements and possibly data, can be departed from in the lower regions of an XML hierarchy, in which elements simply contain data, rather than other elements. The alternative representation for such regions is termed 'Attribute Normal Form' (ANF) and can provide some reduction in verbosity compared to the ENF form of the DTD presented here.[4]

  Figure 9-1 contrasts the ANF and ENF representation of a PVML `location` response. As can be seen the saving is due to ANF not requiring element closing tags – in this instance three such tags (`'filename'`, `'linenumber'` and `'location'`) have been dispensed with.

[4] The DTD presented in Appendix D is, for the sake of clarity, written entirely in ENF but with two exceptions. The `id` attribute (Page 180) requires the expressive power of the `ID` declaration, which can only be applied to an attribute. The `href` attribute (Page 181) is required by the `xinxlude` standard.

```
ENF representation of a location response
<pvml>
  <response>
    <location>
      <filename>
        Test.java
      </filename>
      <linenumber>
        6
      </linenumber>
    </location>
  </response>
</pvml>
ANF representation of a location response
<pvml>
  <response>
    <location filename="Test.java" linenumber="6"/>
  </response>
</pvml>
```

Figure 9-1 Comparison of ENF and ANF representation of a PVML response.

9.2.4 Target Program Input/Output

No mechanism has been proposed for managing the normal input and output of the target program. Where this information is textual a straightforward extension to PVML can manage the entry of data at the engine and the consequent display of program output. Many novice programming scenarios are, quite reasonably, restricted to ones that involve text input and output. For the decoupled engine and target to manage programs with graphical requirements would require a mechanism for a whole additional, complex set of information to be handled. Clearly this is not a matter to be handled by PVML.

The established remote graphical environment is that of X Windows [50]. X Windows can display the graphical output of a target program on any engine running 'X Server' software. The terminology is counter-intuitive – the engine is considered to be offering 'display services' to the host target program. The X Windows approach is very broadly portable but generates a great deal of network traffic. In order to transparently deliver target-

generated graphical output, from programs in several languages, there are several issues that would need to be addressed:

- All target programs would have to perform their graphical output as X-Clients. This will not be generally the case – target programs will use a variety of means to create graphical output. However various UNIX based graphical environments do, in fact, map to an underlying X-Client. Java on UNIX behaves as an X-Client as does the graphical language TCL [81]. The Wine project [124] provides an environment for UNIX that will support the execution of graphical Windows applications by mapping their Windows Graphical Device Interface calls to X-Client requests. Wine can be run under the Windows OS as well.
- The engine applet would need to incorporate X-Server functionality. Various commercial [122] and open source [123] developments support X-Server functionality within Java enabled Web browsers.
- The engine, from the point of view of the novice programmer, would need to manage the inter-relationship of arbitrary target program output windows and the windows that were part of the PV proper.

9.3 Related Work

Previous chapters have examined existing work in two distinct fields that are related to this body of research. This section reviews the relationship of PVML to existing work in the fields of decoupled PV architectures and distributed, language-neutral debugging.

9.3.1 Decoupled PV

As has been stated, the significant impact of the decoupling of target and engine in a PVML-based PV system is that the role of the visualiser can be isolated from the other roles involved in PV. This approach has, most

dramatically, been demonstrated in the work of Roman [90] and Domingue [21]. The Pavane and Vis PV systems both incorporate partitioning of their functionality which leaves a distinct, and independent, location for the activities of the visualiser.

It is instructive to consider how a PVML-based connection would relate to the architecture of these systems and in particular how the dependence of PVML communication on open, 'web friendly', standards would enable such systems to be used through the 'generic' type of Internet connection most commonly encountered by students. These connections are characterised by extensive security-related restrictions that preclude normal, socket-to-socket, communication.

In Pavane, as Figure 9-2 illustrates, it is the communication between the "underlying computation" and the "visualisation computation" that would be realised by PVML. In a functioning Pavane system this stream of program state information is transmitted through inter-process communication mechanisms, using protocols that are particular to Pavane. It would be theoretically feasible to insert a PVML link at this point and hence enable a Pavane visualisation to be viewed at a remote location, through a generic Internet connection.

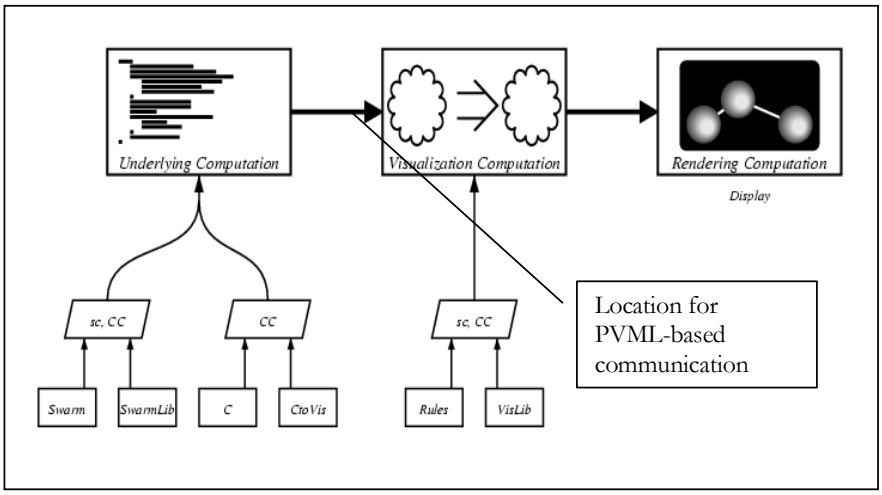

Figure 9-2 Structure of the Pavane system. Reproduced from [91]

In Vis, as Figure 9-3 illustrates, it is the stage at which "program execution history" calls are sent that would be realised in PVML. A similar observation can be made concerning the theoretical application of PVML-based communication to a working Vis system.

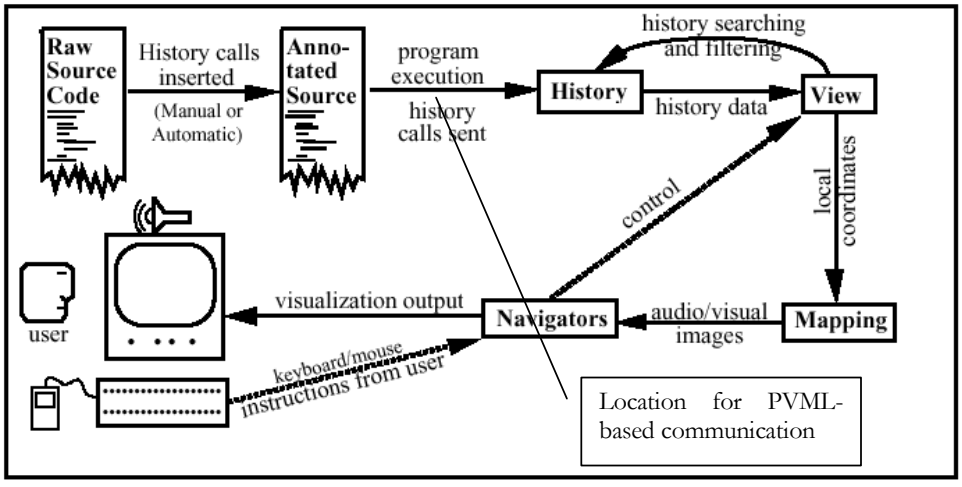

Figure 9-3: The Vis Architecture. Reproduced from [21]

9.3.2 Distributed Debugging

As has been observed in Chapter 5, the content of a PVML stream has much in common with the communication between the components of a distributed debugger. PVML adds the intention that this communication be independent of target programming language to the general requirements of a distributed debugger. In general, as described in Section 5.3, language-neutral, or heterogeneous, debugging has been implemented through a debugging language that abstracts low-level debugging primitives. This indeed, is the approach taken by PVML.

In discussing the relationship of PVML to existing PV systems it has been suggested that, in theory, a target driver could be devised that encapsulated the appropriate components of a PV system in order to communicate program state information remotely. A similar proposition can be made in relation to debugging language implementations – namely that a PVML target driver can be designed to enable a PVML engine to interact with a target built upon an underlying debugging language. The complexity of such a task though would be dependent on the modularity of the design of the debugging-language system under consideration and on the precise architecture of the language implementation. When the boundaries across which the debugging system is decoupled match the boundaries implicit in PVML, as they do in both Pavane and Vis, the task could be considerably more straightforward.

Accordingly the examination of distributed debugging systems, in relation to PVML, is strongly motivated by a consideration of the boundaries across which they are decoupled. In this light, it is the work of Hanson [39] that is of particular interest. The architecture of `deet` and the architecture of a PVML-based system are very similar.

`The deet` program is based on `cdb` [39], earlier work of the same author. The `cdb` program is a machine independent debugger that attaches a small 'nub' of machine dependent code to a target program, in order that machine

dependencies can be abstracted. Figure 9-4 illustrates this architecture and it is instructive to compare this with the PVML architecture shown in Figure 4-4.

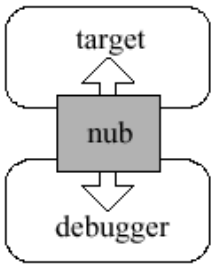

Figure 9-4 `cdb`'s design. Reproduced from [38]

The nub provides a simplified debugging interface, that an external debugger can interact with. In `cdb` the debugger is a text-based debugger, similar to many conventional debuggers.

The `deet` program provides a graphical front-end to the interface provided by an enhanced version of the `cdb` nub. The interface provided by the `deet` nub is shown in Figure 9-5. As can be seen this is a basic, but adequate, set of debugging primitives similar to that provided by GDL [18] and discussed in some detail in Section 6.1.

The principal observation regarding such debugger primitives is that their use can involve significant processing overhead relative to the more expressive commands of an established debugger. A straightforward example of this to contrast the low-level commands required to step forward in execution to the next source line – a part of normal debugger functionality – with the native implementation of a conventional debugger. In a PVML-based scenario, in which a non-trivial network communication is included, this argument carries still more weight.

`deet_open`	initialize the target
`deet_breakpoint f -set j -delete j -list g` *file line character*	set, remove, and list breakpoints
`deet_frame [n]`	get/set current frame
`deet_getval` *type address*	read a value of *type* from *address*
`deet_putval` *type address value*	write the *value* of *type* to *address*
`deet_continue`	resume execution
`deet_sym f -all j -files j -locals j -params | -name` *name* `g`	finds the symbol-table entries
`deet_type` *symbol*	get *symbol*'s type information

Figure 9-5 `deet`'s nub interface. Reproduced from [38]

Nevertheless the approach taken by `deet` shares many aspects with that taken in this research and it would quite definitely be feasible to design a PVML target driver that encapsulates a `deet` nub.

9.4 Further Work

The proposal for an open, XML-based, communication between visualisation engines and targets opens up many fascinating future directions. The general motivation of these developments is to provide a variety of visualisation scenarios in which distinct styles of, and approaches to, visualisation can be employed with a variety of target programming languages. The intention is to open the domain of program visualisation, as a component of introductory programming pedagogy to more extensive experimental evaluation.

Specific developments can be broadly divided between those that develop engine and those that develop target functionality.

9.4.1 Target Development

The extent of PVML that has been demonstrated herein is restricted to a generic core of functionality that can be applied to a cross-section of programming languages. Aside from devising target drivers that encapsulate individual, specific languages and their debuggers there is a specific development that could lead to exploration of the application of PVML to extensive, novel, areas.

.NET Target

The Microsoft .NET framework [61] is a set of standards that enable programs written in a variety of source languages to interoperate. Programs written in wide variety of languages – seventeen non-Microsoft languages are listed at [63] – are compiled into a Common Language Runtime (CLR) that can be executed on a variety of platforms. The interoperability between this multitude of languages is at the CLR level.

The .NET framework incorporates a debugging API [64], that supports the debugging of CLR executions and which gives access to the particular programming language source code that gave rise to each executing fragment of CLR code.

A PVML target driver that encapsulated a CLR debugger would expose the broad cross-section of .NET languages to a PVML engine. The .NET supported languages are representative of the three language paradigms described on page 81, and this target would be a suitable environment for the study of the paradigm-specific extensions to PVML that would support appropriate pedagogy in a variety of CS1 and CS2 environments.

This work has commenced under an honours-level project, supervised by the author.

9.4.2 Engine Development

A general model for PVML engine development is shown in Figure 5-2 where the PVML engine driver is represented as an encapsulation of an existing imperative or declarative visualisation system. As can be seen, a declarative model of visualisation, which is inherently designed to consume a stream of program state information, requires a simpler engine driver.

A general direction for future work on PVML engines would be to implement drivers for a number of existing visualisation front ends. As has been suggested in Section 9.3.2 (page 166), the manageability of this task will depend on the extent and nature of the modularity already demonstrated by the existing PV system. A straight-forward addition of an existing imperative PV system, JSamba, to the reference engine has already been discussed in Section 8.2 (page 130).

Chapter 2, which discusses a novice programming environment, could be the basis for a particular PVML engine which would, in fact, be a complete multi-lingual, novice programming environment. Section 6.8 discusses the extensions to PVML that would support compilation of the target program and simple source file management.

9.4.3 Combined Target and Engine Development

In some cases it could be considered useful to partition an existing, monolithic development or visualisation environment across PVML boundaries. For the novice programmer the effect would be to provide location independence, with a single set of programming tasks being pursued from any location.

BlueJ Target/Engine

In the context of the author's professional involvement in the pedagogy of object-oriented programming the BlueJ [57] environment has been a critical improvement in presenting the paradigm to novices. As discussed in Section

6.5.2, the internal partitioning of the BlueJ implementation is one that would quite naturally support the inclusion of a PVML-based connection between a server-based target and a portable, possibly browser-based, engine.

9.4.4 PVML Development

Undoubtedly the suggestions made for future work would give rise to extensions and possibly modifications to the PVML language proposed herein. The standard definition of the language itself, if such developments are to proceed in an organised fashion, will need to be made available through an appropriate, centralised repository.

CONCLUSION

Program visualisation is a well-established field, populated by a wide variety of systems. These systems demonstrate a range of approaches to providing visual representations of program execution. Many systems have the express purpose of supporting novice programmers in their initial programming endeavours. Programming in a variety of target languages is supported and the visual representations, provided by some form of engine that the user of the system interacts with, involve visual methodologies that in many cases are the express project of the system designer.

The extent of this activity is largely motivated by the suggestion that PV assists the programming novice in forming mental models of an unfamiliar process and will ultimately speed up the development of programming skills. Whilst fundamental to all PV development, this assertion is one that still lacks extensive, empirical support.

Much work has been put into taxonomic analyses of these efforts but, as has been noted, comparatively few researchers have undertaken a concerted, conceptual analysis of what PV actually is. Terms have been defined that identify components and aspects of the PV endeavour but this language has been applied to describing what has been undertaken rather than analysing, through generalised reasoning, the aim of PV. From this point of view the work of Roman is distinctive and the research presented here is profoundly influenced by that work.

Roman's generalised definition of PV as "a mapping from programs to graphical representations" involves a closer than usual examination of the human roles involved in PV. The definition of the roles of user, programmer

visualiser and PV system developer has particular consequences to any discussion of PV systems. Although the user and the programmer role, whose concerns coalesce in the novice-programmer, represent the 'end-user' of a PV system it is the visualiser, who makes the decisions as to exactly how program execution is represented. These choices are central to any assessment or evaluation of PV. In most PV system implementations it is the PV system developer who makes these decisions and there exists no clear location for the independent exercising of the visualiser role.

Isolating the activities of the visualiser, and exposing them to evaluation that is independent of the PV system developer in particular, depends on the design of the PV system itself. A monolithic architecture, in which the PV system consists of a single large program, necessarily involves visualiser decisions being made by the system developer. Roman, in proposing a declarative model of program visualisation, also implied a decomposition of this monolithic architecture such that visualiser activity was expressly isolated.

At one level the contribution of PVML can be expressed in these terms alone. PVML implements a decoupled PV architecture, which echoes that of Roman and several other researchers, but does so in an open and extensible manner. Through PVML it becomes feasible to propose arbitrary assemblies of PV engines and targets and by this means to expose visualiser activity to critical, comparative evaluation. Completely new PV components can be developed or else, as has been described, parts of existing PV systems could be exposed in this manner.

It should also be noted though, that this proposed decoupling of target and engine is precisely the architectural foundation that is needed for a language and location independent programming environment. Historically it is this goal, as expressed in the author's 1999 conference paper, which gave rise to the initial proposal of a Program Visualisation Meta Language. Whilst PVML can provide program visualisation in a fully decoupled environment, it can also provide the elementary program development scaffolding that, almost

inevitably, will accompany a novice programmer making use of PV in their initial programming endeavours. The PVML-based program development environment is an engine that can be used from any location to undertake development of remote target programs – in theory independently of the programming language in which they are written.

The location of PVML within the particular decomposition of the PV task has the interesting consequence that PVML is also a debugging language. This arises because a strict adherence of the proposed PV boundaries to the separation of the PV roles, leads to the PVML stream containing only program state information. The genesis of this definition can be found in PV systems designed by Roman and others, where the activity of visualiser is supported through an un-encumbered stream of program state information. PVML provides such a stream.

It is this architectural consideration that motivates a significant portion of this thesis, leading to an emphasis on the design of debugging languages and attendant low-level programming issues. It leads to the important characterisation of PVML as implementing an abstract debugger that encapsulates a particular concrete debugger in a particular target. Most decisions in the design of PVML can be represented as abstract-to-concrete debugger command mappings.

Although these observations may lead to the suggestion that the name PVML does not accurately describe the work that has been undertaken, on balance it is the attention to the PV domain that has motivated this work. It is the PV domain that stands to benefit principally from adoption of a PVML approach.

This research conclusively demonstrates a loosely coupled, extensible, communication framework through which arbitrary target and engine components can communicate. Not only does this expose program visualisation to substantial opportunities for empirical validation but it also

suggests directions for the significant development of novice programming environments.

Appendix A

LINE-NUMBERED VARIABLES AND SCOPE NAMES IN PVML

This appendix contains a more detailed discussion of the requirement for PVML to identify scopes and program variables through program line-numbers in order to refer to variables in a generic manner.

In all programming languages variable names are unique within a particular, defined, region of the program. The region within which a variable name is considered unique is termed the 'scope' of the variable. The unique identification of a variable can be decomposed into a combination of the variable name and some unique definition of the scope.

The rules relating to scope vary amongst programming languages as some brief examples will show.

Scope in C

Within a program function names must be unique. Within a function variable names must be unique. A C variable is therefore uniquely identified by a function name-variable name tuple. However, as will be seen below, this description of a variable may not be unique in other languages.

Scope in C++

C++ (and some other object oriented languages) allow the overloading of function (method) names. In a language such as C, a function name such as `test()` must be unique within a program. C++ distinguishes between different versions of `test()` according to the types of the parameter(s) declared. Hence `test(int i)` is considered distinct from `test(float f)`. The engine needs to refer to these scopes in a manner that maintains the distinction. Furthermore the scope of a method name is limited by the class in

which it is defined - two distinct classes may include methods of the same name.

The C++ compiler keeps track of this through a process known as 'name mangling', in which methods are given specially generated, unique, names during compilation. These unique names are built by the compiler through a combination of the class name, the method name and some representation of the parameter types. These 'mangled' names are usually private to the compiler but use of appropriate compiler switches, or executing a C++ program under a debugger, can make the mangled names apparent.

From the point of view of a novice programmer, viewing the source code of a C++ program through an engine, variables will be perceived as distinct by virtue of their location in the program source code. A particular variable declaration, as seen in source code listing, will identify a particular variable uniquely. This leads to the requirement that a PVML engine identifies variables through their location in the source file, delegating the retrieval of the actual variable to the target, which also has access to the source file.

In addition, the scoping of variables in C++ is also distinct at the lower level of, otherwise un-named, blocks of source code. Any language construct in C++ that permits the use of braces ('{' '}') to define a block of source code will have the consequence of defining a new scope within which variables may be declared.

The PVML engine requires the ability to refer uniquely to variables declared in each such scope.

Scope in Java

Similar scoping issues arise in Java and these concepts are demonstrated through the fragment of Java source code presented below. In this example

line numbers, not normally part of the Java language, have been added to support the discussion.

```
1  public SampleClass{
2    int j=13;
3    public static void sampleMethod(){
4      int k=14;
5      System.out.println( j );
6      for( int j=0; j<2; j++ ){
7        System.out.println( j );
8      }
9      System.out.println( j );
10   }
11 }
```

Figure A-1: Multiple scopes in a sample Java program

Deleted: 10-1

The fragment of code contains three variables of `int` type but in one case the same name `j` has been used in different scopes.

The variable `j` has been declared in the body of the class (line 2) and assigned a value. The scope of this variable is the entire class. Hence line 5 will print out '13'. In order to place a watch on this variable it might be identified as simply '`j`' in the source file - the syntax of Java dictates that this is an unambiguous reference.

A method named `sampleMethod` is declared in line 3. The effect of this is to establish a new scope in which variables can be declared. The variable `k` has been declared and assigned a value. A representation such as '`sampleMethod, k`' for this variable would not, necessarily, be unique given that `sampleMethod()` may have been overloaded. The requirement is already apparent for a representation that uniquely identifies a particular region of code where a variable has been declared.

This requirement is re-emphasised when, in line 6, yet another scope has been created. The effect, in Java, of using braces after a statement like '`for( j=0…)`' is to establish a scope in which variable names, such as the counter `j`

have the potential to obscure similarly named variables in a superior scope. The effect of this is that line 7 will print out '0' the first time it is executed - rather than '13'. A reference to this particular variable again needs to be made in terms of the specific location where the variable is declared.

Discussion

The requirement that the PVML framework be programming language neutral introduces a particular set of constraints to this discussion. The means employed by the engine to refer to variables and scopes, needs to be independent of particular programming language techniques, such as name mangling. At the same time, the reference to a variable needs to be one that the language-specific target can decode in order to give access to the value of a particular variable.

PVML make use of two terms to identify a variable:

- `filename`

  The source filename in which the variable is declared. The novice programmer will have a 'pretty printed' view of all relevant program source at their disposal and will select a variable by highlighting its declaration.

- `linenumber`

  The line number of the variable declaration. The line number and the source filename will be resolved by the target to reference a particular variable.

This approach assumes that programs are 'well formed' in the sense of using new lines to separate declarations from other source constructs. Handling the program 'conundrums', that deliberately set out to write entire programs in a single line of source code, would need to be set aside as being beyond the scope of this work.

In addition to describing variables PVML targets need, when specifying watched variables, to refer to contexts in the source program by names that identify them uniquely across variable program language contexts. These unique scope names are generated by appending the line number of the beginning of the scope an enclosing scope name.

The sample Java program in Figure A-1 illustrates this concept. This sample Java class contains three distinct scopes and their unique PVML names are shown in Table A-10-1. This table explains the PVML scope names that, accompanied by the filename containing the scope definition, will uniquely identify the respective scopes.

Scope description	PVML scope name	Comment
Entire class	***1	In Java the top-level in a file has a name – in this example `SampleClass`. However in other languages this may not be the case. Hence a general term is used (which cannot be a legal function or method name) to denote this top-level scope.
sampleMethod	sampleMethod3	The PVML scope name describes this scope uniquely, even when the method name has been overloaded.
for{} loop in sampleMethod	sampleMethod6	The PVML scope name identifies a region that would otherwise be anonymous.

Table A-10-1 PVML scope names in a sample Java program

Appendix B

DATA VALUES IN PVML

The purpose of this appendix is to clarify the manner in which arbitrary target data values and references can be encoded in a PVML stream. The hierarchical representation is defined by the DTD presented in Appendix D and this appendix will expand upon the terse representation provided by the DTD.

The root of a data representation in PVML is a `value` element that contains branches and links that represent the structure of target data. A compulsory attribute of all `value` elements is an `id` to which the target driver may assign a target machine memory address.

In the target programming language, data is identified either by the name of a variable or by a pointer that references a region of target memory that is supplying some structured storage. The need to support both data values and data references (as laid out in Section 6.6) means that whenever a data value is transmitted it will be accompanied by a unique target memory reference and, when it exists, a uniquely specified variable name.

Target data representations, encoded in the format described here, may be transmitted in three circumstances:

- Content of an asynchronous request

  These arrive at the engine as a result of watchpoints in the target program being triggered. The debugger causes the target driver to send a PVML request to the engine which will result in visualisation(s) being updated.

- Content of a synchronous response

  These arrive at the engine in response to a request sent to the target such as `read`. Such requests are the result of user interaction with the engine. This will occur when the target program is halted and the user is investigating the state of target data.

- Parameter to a request

  Target data representations will be sent by the engine, as a parameter to a `write` request, when the user is modifying data at the target.

The XML representation is designed to encode hierarchical, and linked, data structures. In all cases the actual data values are ultimately encoded as text or a raw data encapsulation that is manageable by XML and the particular engine implementation.

The XML Protocol Working Group [112], a body undertaking the specification of requirements for the XML Protocol, has largely set aside the issue of the encoding of binary data as being beyond the scope of the XML protocol. Reference is made [ibid Section 2.1], to "commonly used image formats like PNG, JPEG" and to emerging approaches "based on MIME multipart" both of which are in extensive use in related Internet activities.

The components that interoperate to implement the Internet and World Wide Web rely, in many cases, on communicating binary data, in which the entire 8 bits of a byte are significant, through channels that require ASCII formatted data, in which only 7 bits are significant. A typical instance of this constraint exists in electronic mail – the standard defining the format of Internet mail, RFC822 [44] specifies that electronic mail messages must consist of ASCII text. To accommodate this constraint a variety of encodings, such as MIME, have been developed that transform binary data

into an ASCII representation that can be encapsulated in an electronic mail message.

The intent of XML development is to delegate the management of binary data to an accepted, non-XML, standard which, in a PVML context, would be agreed between the target and engine.

Since a PVML target is a wrapper around a debugger for the source language, the data representations and features available through PVML will be constrained by those available in the underlying debugger. The content and format of data appearing in the PVML stream will always be a subset of the representations provided by the debugger involved.

The `data` Element

The value of a target data item, which may consist of arbitrarily nested data structures or references, is transmitted in a `data` element. The `data` element is defined as follows:

Name of variable

Variable names need to uniquely identify a variable, taking into account the scoping rules of the target language. The general form of a variable identification is discussed in Appendix A and consists of:

- `filename`

  The source file name expressed as a target file system location.

- `linenumber`

  The source line number of the variable declaration.

- `varname`

  The name of the variable.

Value of variable

- `value`

  The `value` element can be used recursively to define arbitrarily nested, and linked, data.

The `value` Element

The visualiser in a PV system will be assigning mappings between the state of target program data and, usually visual, representations that are presented to the user. The `value` element in PVML contains the information on which the visualisation will be based.

- `type`

  A text string representing the data type of this variable. This string is only used at the engine for display purposes and will be in the language dependent format employed by the underlying debugger at the target.

- `varname?`

  The "?" syntax in a DTD indicates that the marked item may occur zero or one time in an element – in other words the content is optional.

  `varname` represents the name of the variable. The name is provided in order that the visualisation may incorporate variable names. In certain cases a variable may have no name – for example an intermediate value in a complex expression or a structure that is being referenced through a pointer.

- `val* | value*`

  The DTD syntax of '*' denotes zero-to-many occurrences of the tagged item. The 'I' operator denotes that either of the operands are valid at this level.

A `value` element may consist either of zero or more repetitions of a `val` element or zero or more repetitions of a `value` element. `value` elements incorporate `name` and `type` elements and are therefore appropriate for the representation of structures, objects or named fields within such elements.

`val` elements contain no meta-information and are therefore appropriate for the representation of, possibly repeated, simple elements.

The `val` Element

The `val` element represents a single simple (not structured) data item. This could either be a data value, of the type defined in the enclosing `value` element, or a data reference in the form of a `ptr` element. Both these possibilities are considered in more detail below. Examples are given demonstrating the representation of various data in a variety of source languages.

Data Values

Actual data values (as distinct from data references, which are described below), will consist of nested `value` and `val` elements, with the values of the leaf nodes in the data structures being stored in the `val` elements. A variety of examples of this representation are presented, using C and Java data structures.

C structure

The fragment of C code shows a variable `myStruct` that consists of a structure containing some numeric values and a nested, second, structure. The PVML fragment represents this variable. It should be noted that the meta-data (`type, varname`) in the PVML stream means that this

representation is self-documenting. The PVML stream contains sufficient information for the visualiser to build appropriate visual representations.

```
struct innerStruct{
    int innerInt a;
}
struct sample{
    int x;
    float y;
    struct innerStruct z;
}

struct sample myStruct;
myStruct.x = 10;
myStruct.y = 3.14;
myStruct.innerStruct.a = 42;
```

```
<value>
   <type>struct sample</type>
   <varname>myStruct</varname>
   <value>
       <type>int</type>
       <varname>x</varname>
       <val>10</val>
   </value>
   <value>
       <type>float</type>
       <varname>y</varname>
       <val>3.14</val>
   </value>
   <value>
       <type>struct innerStruct</type>
       <varname>z</varname>
       <value>
           <type>int</type>
           <varname>a</varname>
           <vaL>42</val>
       </value>
   </value>
</value>
```

Figure B-1 A C structure and its PVML representation

C Array

The C fragment in Figure B-2 shows a variable `myArray` that stores a small, one dimensional array of integers. The PVML fragment represents this variable.

```
int myArray[4];
myArray[0]=1;
myArray[1]=2;
myArray[2]=3;
myArray[3]=4;
```

```
<value>
   <type>int</type>
   <varname>myArray</varname>
   <val>1</val>
   <val>2</val>
   <val>3</val>
   <val>4</val>
</value>
```

Figure B-2 A one-dimensional C array and its PVML representation

Figure B-3 shows a limitation of the PVML DTD in representing a two dimensional array. There is insufficient meta-data in the PVML stream to support a two-dimensional visual representation. A visualiser would be constrained to represent the two-dimensional array in one dimension.

To remove this restriction would require use of techniques, similar to those described in Section 6.5.2, to relay type information to the engine independently of data values.

```
int myArray[2][2];
myArray[0][0]=1;
myArray[0][1]=2;
myArray[1][0]=3;
myArray[1][1]=4;
```

```
<value>
   <type>int</type>
   <varname>myArray</varname>
   <val>1</val>
   <val>2</val>
   <val>3</val>
   <val>4</val>
</value>
```

Figure B-3 A two-dimensional C array and its PVML representation

Java Object

The Java fragment shows a variable `myObject`, storing a Java object. The Java object contains a number of fields – one of which is itself an object. Note that some parts of a Java object will be method source code. Representation of these regions is not part of the PVML `value` element. The PVML fragment shows the representation of this variable.

```
class inner{
    int[] a;
}
class outer{
    int x;
    inner y;
}
. . .
outer myObject = new outer();
myObject.x = 42;
myObject.y.a[0] = 1;
myObject.y.a[1] = 2;
```

```
<value>
    <type>class outer</type>
    <varname>myObject</varname>
    <value>
         <type>int</type>
         <varname>x</varname>
         <val>42</val>
    </value>
    <value>
         <type>class inner</type>
         <varname>y</varname>
         <value>
             <type>int</type>
             <varname>a</varname>
             <val>1</val>
             <val>2</val>
         </value>
     </value>
</value>
```

Figure B-4 A Java Object and its PVML representation

Data References

As described in Section 6.6.2, significant data may be stored and retrieved by means of pointers, essentially anonymous references to target memory locations. These memory references, of no direct significance in the engine environment, can be passed back to the target in order to refer to data. Memory references consist of the reference itself (often referred to as a *pointer*) and a means to identify the location being pointed to. In PVML

locations are identified by an `id` element and references to such locations by a `ptr` element.

The `id` attribute

The definition of all `value` elements incorporates a compulsory (`#REQUIRED`) attribute named `id`:

```
33:[5] <!ELEMENT value ( type, varname?, ( eoc | val* |
value*)>

34: <!ATTLIST value id ID #REQUIRED>
```

This attribute is defined in the DTD as being of type `ID`. In XML this implies that the value of the `id` is unique in the document and can also be straightforwardly referred to elsewhere.

The `ptr` element

References (pointers) to memory locations are represented in PVML by a `ptr` element:

```
31: <!ELEMENT ptr (xinc:include, mod? )>
```

This element is an alternative to a raw data value as the form for a `val` element:

```
35: <!ELEMENT val (#PCDATA | ptr)>
```

This element can store what is, in effect, an XML reference to a location in another document in the form of an `xinc: include` element. The xinclude [116] mechanism is defined to support the inclusion, in an XML document, of XML fragments from other documents. This definition of the `ptr` element

[5] The numbers preceding DTD fragments refer to the DTD listing in Appendix D

makes use of a 'namespace prefix', `xinc`[6], in order that downstream XML processors can be directed to handle the element by resolving a reference in another context.

The element `xinc:include`, which is recognised by XML parsers as representing a remote inclusion, incorporates a compulsory (`#REQUIRED`) attribute named `href` through which the location that is being pointed to is specified:

```
59: <!ELEMENT xinc:include EMPTY>

60: <!ATTLIST xinc:include                    href CDATA #REQUIRED>
```

At this stage the expressive power of the DTD format has been exhausted – the `href` attribute is simply defined using the term `CDATA` which is completely generic.

In the context of PVML the `href` parameter needs to define the location of another `value` element – in other words the pointer, points to some data. The general format of such a definition would be:

```
filename#xpointer(id( idvalue ))
```

The keyword `xpointer` means that within the 'file' specified the `filename` a location will be described using XPath [113] syntax. XPath provides an extensive syntax through which sub-sections of an XML document can be defined. In the case of the PVML `ptr`, an extremely restricted subset of XPath is used – namely the `id()` statement, through which an XML node, in a given document, can be identified by an `id`.

[6] The head of the XML document, in other words each PVML fragment, includes the declaration `xmlns:xinc="http://www.w3.org/2001/XInclude"` . The effect of this definition is to force appropriate expansion of XML elements that are preceded by '`xinc:`' // STYLE TO FULL SIZE???

For this style of reference to succeed, the DTD must include an element definition of type `ID` – as has been described in the previous section.

The effect of this syntax is to provide two levels at which data can be pointed to:

- `filename` provides an outer level. The PVML stream, that contains the location being pointed to, is not written to the target or engine file-system at any point. Thus the `filename` field will never contain an actual filename. Instead an `id` value will be used. The target and engine drivers will resolve this reference among stored, top-level, PVML `value` blocks – each of which will contain the compulsory `id` entry.
- `idvalue` provides an inner level of referencing that will function within a `value` block and refer to a `value` element that is nested within the top-level `value` block.

These two levels of representation map directly to the fundamental operations that flow between a PVML target and engine.

A typical user (novice programmer) will select data items whose values are to be monitored in the visualisation. These values may, or may not, include other data values. The PVML `data` request, through which the changed values are returned to the engine, consists of a top-level `value` element along with possible included `value` elements to an arbitrary level of nesting. The entire population of `value` elements, known to the engine, hence falls into these two categories of top-level and subsidiary elements. The decision to preserve this distinction in remote references is based on efficiency considerations.

Pointer Example

This section presents a concrete example of the PVML representation of data references. The scenario presented is the linked list example that has been shown earlier in the general discussion of data representation in procedural languages (Section 6.6.3). For convenience the illustration of the representation of a linked list in DDD is repeated below.

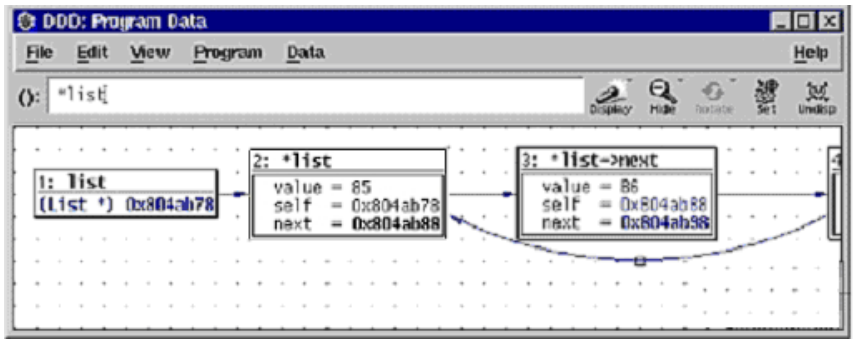

Figure B-5 Data Structure Visualisation in DDD. Reproduced from [28].

The PVML representations of two aspects is presented:

- The first item in the list

  This is labelled '2: *list' in Figure B-5. This data structure includes three members. The value field, which is equal to 85, is assumed to be stored as an 8 byte integer. The self field is a self-referential pointer – the PVML representation of which is omitted for clarity. The next field, a pointer to the next item in the list, is shown as an example of the PVML ptr representation.

- Sub item reference

  The resolution of a `ptr` reference to a sub item in the list – namely an explicit reference to the `next` field in the second list item.

First item in list

The PVML data request shown in Figure B-6, represents the first item in the list, identified as '`2: *list`' in Figure B-5. The visual consequence of this request would be subject to the respresentational decisions of the visualiser role but typically might be similar to that in the DDD screen shown above. This data request would be sent by the target as a result of a request to `watch` this location or possibly a subsequent change in the data value at this location.

The notation used to refer to the type of the pointer is worthy of discussion. The notation '`*list`' is a programming language dependent string that has been provided by the target though interaction with the underlying debugger. In the engine context this is no more that a label that may, at some later point, be passed back to the target.

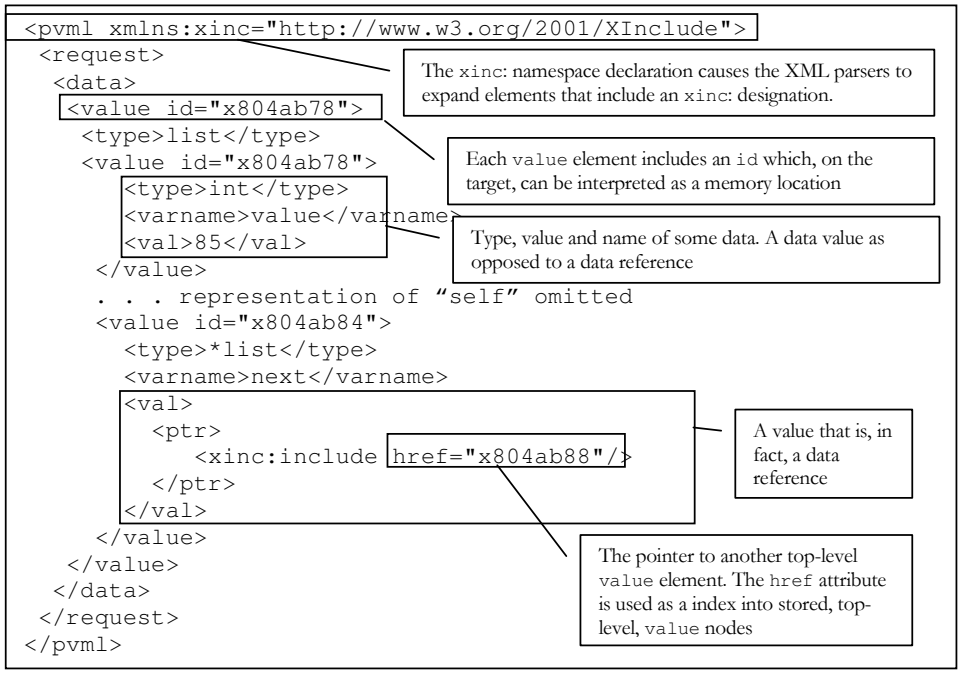

Figure B-6 PVML description of first item in list

Sub-item reference

The PVML data request shown in Figure B-7, represents the `next` field in the second item in the list. This list item is identified as '`3: *list->next`' in Figure B-5 and the PVML shown refers to the `next` field at that location.

As before, the visualiser ultimately would control consequence of this request, but the item being watched in this instance is a single pointer value. The PVML in this example represents what is in effect, a pointer to a pointer and hence is of an appropriate type – '`**list`'. Type names are subject to warning already made concerning their relevance in target and engine.

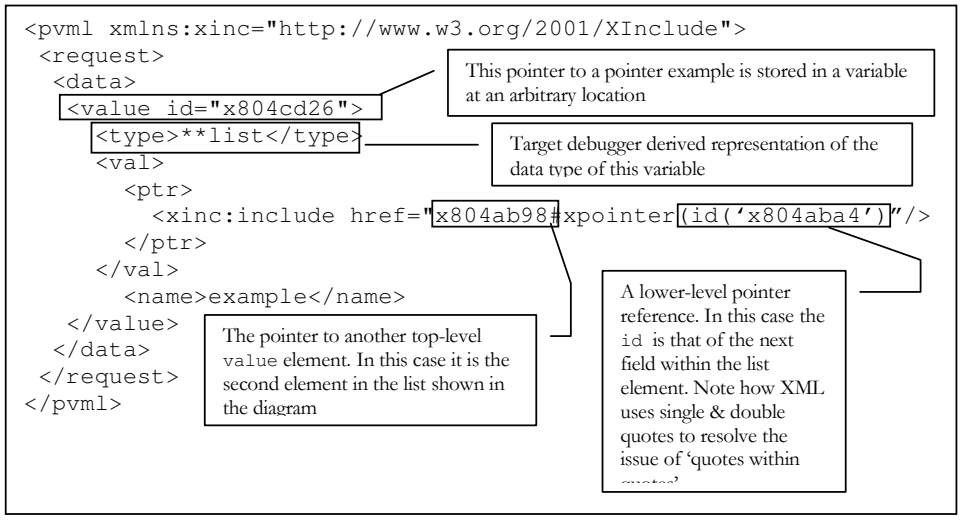

Figure B-7 PVML description of `next` pointer

Pointer Arithmetic

It is common in languages that make use of pointers for operations, known as pointer arithmetic, to be performed on those pointers. The language compiler enforces a view of pointers that preserves their relationship to the storage of data, of the type for which the pointer has been declared. For

example adding '1' to a pointer to a `list` structure does not increment a machine address by '1' – instead the pointer value is incremented by the number of bytes occupied in memory by one additional `list` structure.

Visualisation of a program that includes such pointer arithmetic requires PVML to include a means to describe arbitrary offsets from given pointer values.

To assess the need for PVML to represent pointer arithmetic it is necessary to consider the three circumstances under which PVML data representations are required to be sent, in either direction, between a target and an engine as set out in the introduction to Appendix B.

Content of an asynchronous request

The target is sending updated data to the engine, in this instance as a result of some pointer arithmetic having occurred. This arithmetic has caused changes in the value of data that is already being represented in `ptr` elements.

The updated `href` values will be sent to the engine. It is possible that these new values may not reference data of which the engine currently has a representation. In this case the visualiser must arrange that a `read` request is sent in order to retrieve the required data.

Content of a synchronous response

The engine has requested data from the target with a `read` request and the target is responding. Pointer arithmetic needs to be considered if the visualiser sets out to offer the user functionality that enables them to request to view data at an offset from an existing pointer.

This functionality is provided in PVML by including an optional modifier in the definition of the `ptr` element:

```
31: <!ELEMENT ptr (xinc:include, mod? )>
```

The operation requested though this element is implemented at the target and can be any legal operation supported by the debugger as the example below illustrates.

```
<read>
   <ptr>
       <xinc:include href="x804ab88"/>
       <mod>+7</mod>
   </ptr>
</read>
```

Figure B-8 Reading at an offset from a pointer

Parameter to a request

In this case the engine is making a `write` request in order to modify data in the target program. Similar reasoning applies to the `ptr` element of this request as to the `read` request described above.

Appendix C

SOURCE CODE REPRESENTATION IN PVML

In order to support such features as the pretty printing of source code and source expression stepping the source code must be sent from the target to the engine as a hierarchical structure derived from the parse tree of the program. The ability of the target to provide this information will depend on the extent and type of language support available to the underlying debugger.

There have been three levels of support identified in the process of developing the reference targets against which PVML has been evaluated:

- Level 1

  A level 1 target is unable to provide any hierarchical representation of the source code. In this context the only view of the source code that can be provided will be entirely plain text.

- Level 2

  A level 2 target has access to the program parse tree and hence can deliver a hierarchical representation of the source code as described in this appendix.

- Level 3

  A level 3 target has access to structures that link source expressions to machine code locations. The combination of these associations, and a hierarchical representation that identifies source expressions, is sufficient to support source expressions stepping.

Flat Representation

In order to encode arbitrary text, such as a program listing, in XML a means must be devised to manage the reserved characters of the XML language – such as '>' and '<'. XML provides a mechanism for 'escaping' these specific characters to permit them to passed explicitly in an XML stream but use of this mechanism would require the source code to be parsed at target and engine. The alternative, used for level 1 PVML, is to 'escape' the entire source code as a block using the XML `CDATA` construct.

Hierarchical Representation

The hierarchical representation of PVML used in level 2 & 3 operation makes use of a number of 'complex types' defined in the DTD. These types are described below. The line numbers reference the DTD in Appendix D.

- `32: source ( rawsource | block* | line+ )`

  The code response makes use of a `source` parameter to transmit the program source code and this parameter may consist of a `rawsource` block (in the level 1 case) or else a number of `lines` and `blocks` of code. There must be at least one `line` of code (as defined by the '+') but there may be zero occurrences of `block` ('*')

- `27: block ( line+ )`

  A `block` of source code consists of one or more `lines` of code.

- `30: line ( num, ( expr* | identifier* | literal* | keyword* | tag* | decl* | comment ))`

  A line of source code consists of a line number followed by zero or more occurrences of various syntactic elements. The elements that are considered significant are those that may play some part in subsequent processing at the engine.

- `50: num`

  Source line numbers are enclosed in a `num` element. PVML sends no data at all for blank lines. The engine must regenerate these at display time.

- `29: expr`

  Expressions are tagged, in level 3 PVML, in order that source expression stepping can be supported. A level 3 engine will be able to highlight the currently executing expression based on the regions tagged with `expr`.

- `45: identifier`
  `48: literal`
  `46: keyword`
  `38: comment`

  In the engine special typographical representations that represent distinct syntactic components in the program source code is based on these tags.

- `40: decl`

  Variable declarations are explicitly tagged to assist the engine in determining variable scope. The visual representation of whether or not a particular variable is in a 'watched' state depends of the engine being able to distinguish variables of the same name, in different scopes. The engine can identify variables through the `decl` tag which draws attention to their declaration.

Appendix D

PVML DOCUMENT TYPE DEFINITION

This appendix presents the DTD for PVML. In the interest of clarity the comments in this file have been shown in bold text, though normally a DTD would consist of plain ASCII text. The line numbers have been added to facilitate cross referencing.

```
1  <!--pvml.dtd  Version 0.2 Supports watching of variables-->
2  <!ELEMENT pvml ( request | response ) >
3  <!ELEMENT request ( run| next | step | cont | break | list | query |
       watch | save | file | compile | frame | data ) >
4  <!ELEMENT response ( code | location | breakresp | pvmlinfo )>
5  <!--Requests - To target -->
6  <!ELEMENT break ( filename, linenumber )>
7  <!ELEMENT cont EMPTY >
8  <!ELEMENT file ( path )>
9  <!ELEMENT list ( filename )>
10 <!ELEMENT next EMPTY >
11 <!ELEMENT query EMPTY >
12 <!ELEMENT read ((filename, linenumber, varname)|ptr)>
13 <!ELEMENT run ( appname )>
14 <!ELEMENT save ( pathname, source )>
15 <!ELEMENT step (numstep?) >
16 <!ELEMENT watch (stat, ((filename, linenumber, varname)| ptr))>
17 <!ELEMENT write (((filename, linenumber, varname)| ptr), value)>
18 <!--Requests - From target -->
19 <!ELEMENT data (  filename, linenumber, varname, value )>
20 <!ELEMENT frame ( direction )>
21 <!--Responses - From target -->
22 <!ELEMENT code ( pvmllevel, filename, source )>
23 <!ELEMENT breakresp ( set )>
24 <!ELEMENT location ( filename, linenumber )>
25 <!ELEMENT pvmlinfo ( debugger )>
26 <!--Complex types -->
27 <!ELEMENT block ( line+ ) >
28 <!ELEMENT decl ( identifier | )>
29 <!ELEMENT expr ( identifier* | literal* | keyword* | tag* )>
30 <!ELEMENT line ( num, ( expr* | identifier* | literal* | keyword* |
       tag* | decl* | comment ))>
31 <!ELEMENT ptr (xinc:include, mod? )>
32 <!ELEMENT source ( rawsource | block* | line+ )>
33 <!ELEMENT value ( type, varname?, ( eoc | val* | value*)>
34 <!ATTLIST value id ID #REQUIRED>
35 <!ELEMENT val (#PCDATA | ptr)>
```

Continued on Page 203

```
36 <!--Basic elements -->
37 <!ELEMENT appname (#PCDATA) >
38 <!ELEMENT comment (#PCDATA) >
39 <!ELEMENT debugger (#PCDATA) >
40 <!ELEMENT decl (#PCDATA) >
41 <!ELEMENT direction (#PCDATA)>
42 <!ELEMENT eoc (#PCDATA)>
43 <!ELEMENT filename (#PCDATA) >
44 <!ELEMENT id (#PCDATA)>
45 <!ELEMENT identifier (#PCDATA) >
46 <!ELEMENT keyword (#PCDATA) >
47 <!ELEMENT linenumber (#PCDATA) >
48 <!ELEMENT literal (#PCDATA) >
49 <!ELEMENT mod (#PCDATA) >
50 <!ELEMENT num (#PCDATA) >
51 <!ELEMENT numstep (#PCDATA) >
52 <!ELEMENT pvmllevel (#PCDATA) >
53 <!ELEMENT rawsource (#PCDATA)>
54 <!ELEMENT set (#PCDATA) >
55 <!ELEMENT stat (#PCDATA) >
56 <!ELEMENT tag (#PCDATA) >
57 <!ELEMENT type(#PCDATA)>
58 <!ELEMENT varname (#PCDATA)>
59 <!ELEMENT xinc:include EMPTY>
60 <!ATTLIST xinc:include href CDATA #REQUIRED>
```

Appendix E

XML PARSERS

There are two fundamentally distinct approaches to parsing XML documents – the Simple API for XML (SAX) and the Document Object Model (DOM). This appendix describes the difference in these two approaches in order to inform the discussion of parsing the PVML stream.

Simple API for XML (SAX)

SAX [114] is described as an *event-based* API and treats an XML document as a stream of text. As the stream of text is consumed, starting at the beginning of the document, the SAX libraries generate a series of *events* that an application program can receive and process. These events correspond, for example, to the opening and the closing of tags in the document. A document, which could be arbitrarily large, is seen by the application program as a series of events and there is no necessity to store the entire document in any internal structures of the program.

This is considered the main advantage of SAX. If the task being implemented is one that does not require access to the entire structure of the document, for example searching for a particular element, then it could be considered an unwarranted overhead to build up a complete description of the document structure within the application. The PVML stream cannot usefully be processed sequentially in this way. The persistent representation, in the engine, of target source and data needs to be in a form that is not far removed from the hierarchical XML structure so that the discarding of that structure, that is fundamental to SAX parsing, would be a negative feature.

A positive feature of SAX parsing however is the event-based architecture which allows access, through programmer supplied routines, to the lowest levels of processing of the source document.

Document Object Model API (DOM)

The DOM [117] API for XML reads an entire document into memory before exposing the XML document to programmatic manipulation. With the entire, hierarchical, structure of the document available to the application the range of manipulations that can be supported is greatly extended. DOM will permit any section of the document to be viewed and even deleted or modified.

The engine can make good use of DOM based representations in order to store and manipulate the PVML transmitted representations of the program source code and regions of data.

A drawback to using the DOM parser is that there is no mechanism available in this framework through which the programmer can over-ride the default, low-level processing. A DOM parser will succeed completely or else fail to parse some region of its input. There is no provision for modifying the low level behaviour of the parser on an element by element basis.

Combined Parsing

The descriptions above of SAX and DOM parsing clearly identify positive aspects of both. On the one hand the output of a DOM parser is useful for subsequent manipulations whilst on the other hand a SAX parser exposes low-level element parsing to programmatic intervention.

The PVML parser uses an approach that can benefit from both these features. The requirement to proceed in this way arises due to the inability

of the DOM parsing routines to sensibly manage empty nodes in an XML document. An empty node can arise when an additional newline character is inserted into an output stream. The DOM parsing routines, even if configured to ignore whitespace, treat this as an additional node in the output tree. The result is a DOM tree that is semantically correct but very difficult to work with due to the number of addition, empty nodes.

A SAX parser can, through a programmer-provided implementation of the `characters()` routine (the event handler that is called by the SAX parser for each group of characters) sanitise the input, removing any empty node definitions.

This event handler can be written in such a way that the sanitised output is written into a DOM tree. Figure E-1 shows the characters() handler that has been provided in the PVML parser. The variable `db` in this routine represents the DOM tree representation that is progressively being built. The routine, that is called for each additional block of characters that the SAX parser sees, uses the variable `stripNewLine` to control the removal of extra newline characters from the data that is written to the DOM. The resulting DOM is guaranteed to be clear of any empty nodes.

```
public void characters (char ch[], int start, int length)
    throws SAXException{
       int stripNewline = 0;
       if( length >= 1 && ch[start] == '\n' )
           stripNewline |= 1;
       if( length > 1 && ch[start+length-1] == '\n' )
           stripNewline |= 2;
       switch( stripNewline ){
          case 0: //No newlines
             db.characters( ch, start, length );
             break;
          case 1: //Newline at start
             if( length != 1 )
                db.characters( ch, start+1, length-1 );
             break;
          case 2: //Newline at end
             db.characters( ch, start, length -1 );
             break;
          case 3: //Newline at start and end
             db.characters( ch, start=1, length -2 );
             break;
       }
    }
```

Deleted: 10-9

Figure E-1 SAX Parser `character()` handler

BIBLIOGRAPHY

1. Anderson, J. and Reiser, B. (1985). "The LISP Tutor" *Byte*, 10(4) pp 159-175.
2. Armenise, P., Bandinelli, S,.Ghezzi, C. and Morzenti, A. (1992). Software Process Languages: Survey and Assessment, In *Proceedings of the Fourth Conference on Software Engineering and Knowledge Engineering*, Capri, Italy.
3. Ashby, G., Salmonson, L. and Heilman, R. (1973). Design of an interactive debugger for FORTRAN:Mantis, *Software-Practice and Experience* 3(1), pp 65-74.
4. Baeker, R. and Sherman, D. (1981). Sorting out Sorting, *SIGRAPH Video Review* 7.
5. Baeker, R.M. and Marcus, A. (1990). *Human Factors and Typography for More Readable Programs,* ACM Press, Addison-Wesley, Reading, Mass., USA.
6. Baskerville, D.B. (1985). Graphic Presentation of Data Structures in the DBX Debugger, *Technical Report UCB/CSD 86/260,* University of California, Berkeley, CA, USA.
7. Bentley, J. and Kernighan, B. (1987). A System for Algorithm Animation: Tutorial and User Manual, *Computing Science Technical Report 132,* AT&T Bell Laboratories.
8. Borning A. (1981). The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation; Laboratory, *ACM Transactions on Programming Languages and Systems* 3(4), pp 353-387.
9. Borning, A., Freeman-Benson, B. and Wilson, M. (1992). Constraint Hierarchies, *Lisp and Symbolic Computation* 5, pp 223-270.
10. Boroni, C.M., Goosey, F.W., Grinder, M. and Ross, R.A. (1998). A Paradigm Shift! The Internet, the Web, Browsers, Java, and the future of Computer Science Education, *SIGCSE Bulletin,* 30(2), pp 145-152.
11. Boroni, C.M., Goosey, F.W., Grinder, M., .Rockford J. and Wissenbach, P. (1997). WebLab! A Universal and Interactive Teaching, Learning, and Laboratory Environment for the WWW, *SIGCSE Bulletin,* 29(1), pp 199-203.
12. Boudier, G., Gallo, F., Minot, R. and Thomas, I. (1989). An Overview of PCTE and PCTE+, *ACM SIGSOFT Software Engineering Notes,* 13(5), pp 248-257.

13. Bray, T., Paoli, J., and Sperberg-McQueen, C.M. (1998). Extensible Markup Language, http://www.w3.org/TR/1998/REC-xml, Accessed 28/3/2003.

14. Brown, M.H. (1988). Exploring Algorithms using Balsa-II, *IEEE Computer,* 21(5), pp 14-36.

15. Brown, M.H. (1988). Perspectives on Algorithm Animation, In *Proceedings of the CHI '88 conference on Human Factors in Computing Systems, ACM Press, New York,* pp 33-38.

16. Brown, M.H. (1991). Zeus, A System for Algorithm Animation and Multi-View Editing, *In Proceedings of IEEE Workshop on Visual Languages*, New York: IEEE Computer Society Press, pp 4-9.

17. Brown, M.H. (1992). A System for Algorithm Animation, *Computer Graphics,* 18(3), pp177-186.

18. Crawford, R. H , Olsson, R. A. , Ho, W. W. and Wee, C. E. (1995). Semantic issues in the design of languages for debugging, *Computer Languages,* 21(1), pp 17-37.

19. Cunningham, W., and Beck, K (1986). A Diagram for Object-Oriented Programs, In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'86)*, Portland, Oregon, USA, pp 361-367.

20. Di Maio, A., Ceri, C. and Crespi Reghizzi, S (1985). Execution monitoring and debugging tool for Ada using relational algebra, *ACM SIGAda Ada Letters,* V(2), pp 109-123.

21. Domingue, J., Price, B.A. .and Eisenstadt, M., (1992). A Framework for Describing and Implementing Software Visualization Systems. In *Proceedings of Graphics Interface 92 Conference*, May 1992, Vancouver, Canada, Canadian Information Processing Society, Toronto, Canada, pp 53-60.

22. Duisberg, R.A. (1986). Constraint Based Animation: Temporal Constraints in the Animus System, *UW CSE Technical Report,86-09*-01, University of Washington, Computer Science & Engineering, Seattle, WA, USA.

23. Eisenstadt, M. and Brayshaw, M. (1988). The Transparent Prolog Machine (TPM); an execution model and graphical debugger for logic programming, *Journal of Logic Programming* 5(4), pp. 1-66.

24. Eisenstadt, M., Domingue, J., Rajan, T. and Motta, E. (1990). Visual Knowledge Engineering, *IEEE Transactions on Software Engineering, Special Issue on Visual Programming* 16(10), pp 1164-1177.

25. Ellshof, I.J.P. (1989). A distributed debugger for Amoeba. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on*

Parallel and Distributed Debugging, May 5-6, 1988, University of Wisconsin, Madison, Wisconsin. ACM SIGPLAN Notices 24(1), January 1989, pp 1-10

26. Emmerlich, W., Mascolo, M. and Finkelstein, A. (2000). Implementing Incremental Code Migration with XML, In *Proceedings of. 22nd International Conference on Software Engineering (ICSE2000)*, Limerick, Ireland, June 2000. ACM Press pp 397-406.

27. Free Software Foundation (2003). The GNU Compiler Collection, GNU Project - Free Software Foundation (FSF), http://gcc.gnu.org. Accessed 16/3/2003.

28. Free Software Foundation (2000). Displaying Data, Debugging with DDD, http://www.gnu.org/manual/ddd/html_mono/#Displaying%20Values. Accessed 20/3/2003.

29. Free Software Foundation (2002). DDD - Data Display Debugger, GNU Project - Free Software Foundation (FSF), http://www.gnu.org/software/ddd. Accessed 20/3/2003.

30. Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley.

31. Garlan, D. and Ilias, E. (1990). Low-cost, adaptable tool integration policies for integrated environments, *ACM SIGSOFT Software Engineering Notes,* 15(6) pp. 1-10.

32. Golan, M., Hanson, D.R. (1993). DUEL - A Very High-Level Debugging Language. In *Proceedings of USENIX Winter Conference*, San Diego, USA, pp 107-117.

33. Goldberg, A. (1994). *Smalltalk-80; the Interactive Programming Environment*, Addison-Wesley, Reading, Mass., USA.

34. Goldenson, D.R. (1989). The Impact of Structure Editing on Introductory Computer Science Education, *ACM SIGCSE Bulletin,* 21(3), pp 26-29.

35. Grisham, R. (1971). Criteria for a debugging language. In *Debugging Techniques in Large Systems*, Ed. R. Rustin, Prentice Hall, pp 57-75.

36. Haajanen, J., Pesonius, M., Sutinen, E., Tarhio, J., Terasvirta, T. and Vanninen, P. (1977). Animation of user algorithms on the Web, *Proceedings Visual Languages '97, IEEE Symposium on Visual Languages*, IEEE 1997, pp 360-367.

37. Haibt, L.M. (1977). A Program to Draw Multi-Level Flowcharts. In *Proceedings of the Western Joint Computer Conference*, San Fransisco, 1959, pp 131-137.

38. Hanson, D.R. and Korn,J.L (1977). A simple and extensible graphical debugger. In *Proceedings of the USENIX Annual Technical Conference*, January 1997, Anaheim, California pp 163-174.

39. Hanson, D.R. and Raghavachari, M. (1996). A Machine-Independent Debugger, *Software—Practice and Experience,* 26 (11), pp 1277-1299.

40. Helttula, E., Hyrskykari, A. and Raiha, K.-J. (1989). Graphical Specification of Algorithm Animations with ALADDIN. In *Proceedings of the 22nd Hawaii International Conference on System Sciences,* January 1989, Kailua-Kona, Hawaii,. pp 892-901.

41. Hendrix, D., Barowski, L. and Cross, J. (1997). A Visual Development Environment for Multi-Lingual Curricula, *SIGCSE Bulletin*, 29(1), pp 22-24.

42. Henry, R.R.,Whaley, K.M. and Forstall, B. (1990) The University of Washington Illustrating Compiler, *SIGPLAN Notices*, 25(6), pp 223-233.

43. Hyrskyakari, A. and Raiha, K.J. (1987). Animation of algorithms without programming. In *Proceedings 1987 Workshop on Visual Languages, IEEE Computer Society*, Linkoping, Sweden, pp. 40-54.

44. Internet Engineering Steering Group (1982). Standard For The Format Of ARPA Internet Text Messages, http://www.rfc.net/rfc822.html. Accessed 30/3/2003

45. JavaSoft. (1999). Java Remote Method Invocation (RMI), http://java.sun.com/products/jdk/1.2/docs/guide/rmi. Accessed 12/1/2003.

46. Jimenez-Peris, R., Pareja-Flores, C., Patino-Martinez, M. and Valazquez-Iturbide, J.A. (1996). Graphical Visualization of the Evaluation of Functional Programs. In *Proceedings of the ACM ITiCSE Conference*. June 1996, Barcelona, Spain, pp 36-38.

47. Jimenez-Peris, R., Patino-Martinez, M. and Velazquez-Iturbide, J.A. (2000). Towards Truly Educational Programming Environments. In *Computer Science Education in the 21st Century*, Ed T.Greening, Springer-Verlag , pp 81-112.

48. Johnson, M.S. (1977). The Design of a High-Level Language-Independent Symbolic Debugging System. In *Proceedings of the ACM Annual Conference*, October 1977, Seattle, WA, USA, pp 315-322.

49. Johnson, M.S. (1982). Some requirements for architectural support of software debugging. In *Proceedings of the first international*

symposium on Architectural support for programming languages and operating systems, March 1982, Palo Alto, California, USA, pp 140-148.

50. Jones, O. (1988). *Introduction to the X Window System*, Prentice Hall Professional Technical Reference.

51. Jonson, W. E. and Soloway, E. (1985). PROUST: Knowledge-Based Program Understanding, *IEEE Transactions. on Software Engineering*, 11(3), pp 11-19.

52. JTB, "JTB: The Java Tree Builder Homepage" in http://www.cs.purdue.edu/jtb/ (2003)???

53. Karlund, N., Moller, A. and Schwartzbach, M.I. (2000). DSD: A Schema Language for XML. In *Proceedings 3rd ACM Workshop on Formal Methods in Software Practice*, 2000, Portland, OR, USA, pp 101-111.

54. Kernighan, B.W and Pike, R. (1984). *The UNIX Programming Environment*, Prentice-Hall Inc.

55. Kernighan, B.W. and Ritchie, D.M. (1978). *The C Programming Language*, Prentice-Hall Inc, p24

56. Knuth, D.E. (1984). Literate Programming, *Computing,* 27(2), pp 97-111

57. Kolling, M. and Rosenberg , J. (1996). An Object Oriented Program Development Environment for the first programming course. In *Proceedings of the 27th SIGCSE Technical Symposium on Computer Science Education*, February, 1996, Philadelphia, PA, USA, pp 83-87.

58. Mancoridis, S., Holt, R. and Penny, D. (1993). A Curriculum-Cycle Environment for teaching programming. In *Proceedings of the twenty-fourth SIGCSE technical symposium on Computer science education,* February 1993, Indianapolis, Indiana, USA, pp15-19.

59. Mattern,F. and Sturm, P. (2003). From Distributed Systems to Ubiquitous Computing - The State of the Art, Trends, and Prospects of Future Networked Systems. In *Proceedings of KIVS 2003 (Kommunikation in Verteilten Systemen)*, February 2003, Leipzig, Germany, Springer-Verlag, pp 3-25.

60. McDowell, C.E., Helmbold , D.P. (1989). Debugging concurrent programs, *ACM Computing Surveys (CSUR),* 21(4) pp 593-622.

61. Microsoft (2003). .NET Framework Home Page, http://msdn.microsoft.com/netframework/. Accessed 6/4/2003.

62. Microsoft (2003). COM+ Reference, MSDN Library, http://msdn.microsoft.com/library/en-us/cossdk/htm/cosreftoplevel_65r9.asp. Accessed 15/3/2003

63. Microsoft (2003). Programming Language Partners, MSDN Library, http://msdn.microsoft.com/vstudio/partners/language/default.asp. Accessed 6/4/2003.

64. Microsoft (2003). Visual Studio Debugger Object Model, MSDN Library, http://msdn.microsoft.com/library/default.asp?url=/library/ en-us/vsdebugext/html/vxoriDebuggerObjectModel.asp. Accessed 6/4/2003

65. Mukherjea, S. and Stasko, J. (1994). Toward Visual Debugging: Integrating Algorithm Animation Capabilities within a Source Level Debugger, *ACM Transactions on Human Computer Interaction,* 1(3), pp 215-344.

66. Mulholland, P. (1997). Using a fine-grained comparative evaluation technique to understand and design software visualization tools. In *Empirical Studies of Programmers: Seventh Workshop*, New York: ACM Press, pp 91-108 .

67. Myers, B.A. (1983). Incense: A System for Displaying Data Structures, *ACM SIGRAPH*, 17(3), pp 115-125.

68. Myers, B.A. (1986). Visual Programming, Programming by Example, and Program Visualisation: A Taxonomy. In *Proceedings, CHI '86: Human Factors in Computing Systems*, 1986, Boston, MA, pp 59-66.

69. Myers, B.A., Chandhok, R. and Sareen, A. (1988). Automatic data visulization for novice Pascal programmers. In *Proceedings of IEEE Workshop on Visual Languages*, October 1988, Pittsburgh, PA, USA, pp 192-198

70. Myers, B.A., Miller, R.C., McDonald, R. and Ferrency, A. (1996). Easily adding Animations to Interfaces Using constraints. In *Proceedings of the ACM SIGGRAPH Symposium*, Seattle, WA, pp. 119-128.

71. Naps, T., et al. (1997). Using the WWW as the delivery mechanism for interactive, visualization-based instructional modules, *Report of the ITiCSE '97 working group on visualization*, ITiCSE-WGRSP '97, pp. 31-26

72. Norman D.A. (1983). Some observations on mental models. In D. Gentner, A. Stevens (eds.) *Mental Models*, Lawrence Erlbaum Associates, Hillsdale NJ, USA, pp 7-14.

73. Notkin, D. (1988). The Relationship Between Software Development Environments and the Software Process. In *Proceedings of the third ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments,*1988 , Boston, MA, USA, pp 107-109.

74. OASIS (2001). Bioinformatic Sequence Markup Language (BSML), Organization for the Advancement of Structured Information Standards, http://xml.coverpages.org/bsml.html. Accessed 2/4/2003.

75. OASIS (2002). Chess Markup Language (ChessML), Organization for the Advancement of Structured Information Standards, http://xml.coverpages.org/chessML.html. Accessed 2/4/2003.

76. OASIS (2003). XML Applications and Initiatives, Organization for the Advancement of Structured Information Standards, http://xml.coverpages.org/xmlApplications.html. Accessed 2/4/2003.

77. OASIS, (2001). Taxonomic Markup Language, Organization for the Advancement of Structured Information Standards, http://xml.coverpages.org/taxonomicML.html. Accessed 2/4/2003

78. Olsson, R.A.; Crawford, R.H.; Ho, W.W.; Wee, C.E. (1991). Sequential debugging at a high level of abstraction, *IEEE Software*, 8(3), pp 27-36.

79. OMG (1997). CORBA™/IIOP™ Specification, OMG Documents, http://www.omg.org/technology/documents/formal/corba_iiop.htm. Accessed 1/2/2003.

80. OMG (2002). XML Metadata Interchange, OMG Documents, http://www.omg.org/technology/documents/formal/xmi.htm. Accessed 2/4/2003.

81. Ousterhout ,J.K. (1990). Tcl: An embeddable command language. In *Proceedings Winter USENIX Conference*, Berkeley, CA, USA, pp 133-146.

82. Price, B.A. and Baecker, R.M. (1991). The Automatic Animation of Concurrent Programs. In *Proceedings of International Workshop on Human Computer Interaction*, 1991, Moscow, USSR, pp. 128-137

83. Price, B.A., Baecker, R.M. and Small, I.S. (1993). A Principled Taxonomy of Software Visualisation, *Visual Languages in Computing* 4(3), pp 211-266

84. Ramsey, N. and Hanson, D.R. (1992). A retargetable debugger. In *Proceedings of the 5th ACM SIGPLAN conference on*

Programming language design and implementation, San Francisco, CA, USA, pp 22-31.

85. Redhat, (2001). Insight Home Page -The GDB GUI, http://sources.redhat.com/insight. Accessed 21/1/2003

86. Rich, C. and Waters, R.C. (1987). The Programmer's Apprentice Project: A Research Overview, *IEEE Computer*, 21(11), pp 10-25.

87. Roman, G.C. and Cox, K. (1989). A Declarative Approach to Visualising Concurrent Program Execution, *Computer,*22(10), pp 25-36.

88. Roman, G.C. and Cox, K. (1993). A Taxonomy of Program Visualisation Systems, *Computer,* 26(12), pp 11-24.

89. Roman, G.C. and Cox, K.C. (1992). Program Visualization: The Art of Mapping Programs to Pictures. In *Proceedings of the 14th international conference on Software engineering*, May 1992, Melbourne, Australia, pp.412-420.

90. Roman, G.C., Cox, K., Wilcox, C. and Plun, J. (1992). Pavane: A System for Declarative Visualisation of Concurrent Computations, *Visual Languages and Computing,* 3(1), pp 161-193.

91. Roman, G.C., Cox, K., Wilcox, C. and Plun, J. (1992). Pavane: A System for Declarative Visualisation of Concurrent Computations, *Technical Report WUCS-92-40,* Department of Computing Science, Washington University, Saint Louis, MO, USA.

92. Rutherford, A. and Wilson J.R. (1991). Models of Mental Models: An Ergonomist Psychologist Dialogue. In D. Ackerman and M. Tauber (eds.), *Mental Models In Human-Computer Interaction,*. Amsterdam: North-Holland.

93. Sandewall, E. (1978). Programming in an Interactive Environment: The LISP Experience" in *ACM Computing Surveys,* 10(1), pp 35-71.

94. Scanlan, D. A. (1989). Structured flowcharts outperform pseudo code: an experimental comparison, *IEEE Software,*. 6(5), pp 28-36.

95. Smith, P. A. and Webb, G. I. (2000). The Efficacy of a Low-Level Program Visualisation Tool for Teaching Programming Concepts to Novice C Programmers, *Journal of Educational Computing Research*, 22(2), pp 187-215 .

96. Sosic, R., Abramson D. (1997). Guard: A Relative Debugger, *Software - Practice and Experience*, 27 (2), pp 185-206

97. Sparud, J., Nilsson, H. (1995). The Architecture of a Debugger for Lazy Functional Languages. In *Proceedings of AADEBUG'95,*

2nd International Workshop on Automated and Algorithmic Debugging, May 1995, Saint-Malo, France, pp 19-34.

98. Stal, M. (2002). Web services: beyond component-based computing, *Communications of the ACM*, 45 (10), pp 71-76.

99. Stallman, R.M., Pesch, R.H. (1991). Using GDB: A guide to the GNU source-level debugger, *Technical report*, Free Software Foundation, Cambridge, MA, USA.

100. Stasko, J. and Patterson,C. (1992). Understanding and Characterizing Software Visualization Systems. In *Proceedings of IEEE Workshop on Visual Languages*, Seattle, WA, USA, pp 2-10.

101. Stasko, J., Badre, A. and Lewis, C. (1993). Do algorithm animations assist learning?: an empirical study and analysis. In *Proceedings of CHI '93. Conference on Human factors in computing systems*, January 1993, Amsterdam, The Netherlands, pp 61-66

102. Stasko, J.T. (1988). JSAMBA -- Java version of the SAMBA Animation Program, http://www.cc.gatech.edu/gvu/softviz/algoanim/jsamba. Accessed 2/4/2003.

103. Stasko, J.T. (1990). Tango: A Framework and System for Algorithm Animation, *IEEE Computer,* 23(9), pp 27-39.

104. Stasko, J.T. (1997). Using Student-built Algorithm Animations as Learning Aids, *ACM SIGCSE Bulletin,* 29(1), pp 25-29.

105. Steven, J., Chandra, P., Fleck, P. and Podgurski,.A. (2000). jRapture: A Capture/Replay tool for observation-based testing. In *Proceedings, International Symposium on Software Testing and Analysis*, Portland, OR, USA, pp 158-167.

106. Stratton, D.H. (1999). Towards a Language and Location Independent Novice Programming Environment. In *Proceedings of the International Conference on Computers in Education (ICCE)*, November 1999, Tokyo, Japan, pp 59-66

107. Stratton, D.H. (2001). A Program Visualisation Meta-Language Proposal. In *Proceedings of the International Conference on Computers in Education (ICCE)*, November 2001, Seoul, Korea, pp 602-609.

108. Sun Microsystems (1985). RPC reference manual, Sun Microsystems Ltd., Mountain View, California, USA.

109. Sun Microsystems (2000). The Java Platform Debugging Architecture, http://java.sun.com/j2se/1.3/docs/guide/jpda/jpda.html. Accessed 13/1/2003.

110. Touretzky, D.S. and Lee, P. (1992). Visualizing Evaluation in Applicative Languages, *Communications of the ACM,* 35(10 0, pp 49-59.

111. UCLA (2002). JavaCC Grammar Repository, http://www.cobase.cs.ucla.edu/pub/javacc. Accessed 24/2/2003.

112. W3C (2000). XML Protocol Working Group Charter, W3C, http://www.w3.org/2000/09/XML-Protocol-Charter. Accessed 24/2/2003.

113. W3C (2002). XML Path Language (XPath) 2.0, W3C Working Draft, http://www.w3.org/TR/xpath20. Accessed 24/2/2003.

114. W3C (2002). Simple API for XML (SAX), http://www.saxproject.org, Accessed 13/2/2003.

115. W3C (2002). Web Services Activity, W3C Architecture Domain Activity Statement, http://www.w3.org/2002/ws/Activity. Accessed 24/2/2003.

116. W3C (2002). XML Inclusions (XInclude) Version 1.0, W3C Candidate Recommendation, http://www.w3.org/TR/xinclude. Accessed 24/2/2003.

117. W3C (2003). The Document Object Model (DOM), W3C Architecture Domain, http://www.w3.org/DOM. Accessed 24/2/2003.

118. W3C (2003). XML Schema, W3C Architecture Domain, http://www.w3.org/XML/Schema. Accessed 24/2/2003.

119. Waters, R.C. (1988). Program Translation via Abstraction and Reimplementation, *IEEE Transactions on Software Engineering*, 14(8*)*, pp 1207-1229.

120. Watt, D.A. (1990). *Programming Languages: Concepts and Paradigms*, Prentice Hall

121. WebGain (2000). WebGain Products : JavaCC, http://www.webgain.com/products/java_cc. Accessed 2/4/2003

122. *WebTerm* (2003). WebTerm X and X Windows, http*:*//www.powerlan-usa.com/webtermx.html. Accessed 2/4/2003

123. *WierdX* (2001). WeirdX -- Pure Java X Window System Server under GPL, http://www.jcraft.com/weirdx/index.html. Accessed 2/4/2003

124. Wine (2003). Wine Development HQ, http://www.winehq.com. Accessed 2/4/2003