INFORMATION TECHNOLOGY

114 Surviving Paradigm Shifts in Software Development Technology: A Management Case Study of Industrial Experience
C CONNAUGHTON, CNG DAMPNEY

124 SortEAM: A Design History and Justification Maintenance Tool
GT JAYAPUTERA, KE CHENG

134 Software Reusability in Australia
R VERSCHOOR, G LOW

143 SEE-Ada: Software Evaluation Environment for Ada
G SLADE, N PARKER

151 Generic Re-Engineering Environment Design Criteria: An Evaluation of the Software Refinery™
PA BAILLES, M CHAPMAN, M CHIA, I PEAKE

158 Graph-based Class Testing
D HOFFMAN, P STROOPER

SPECIAL FEATURES

113 Editorial

164 Book Reviews
The Australian Computer Journal is an official publication of the Australian Computer Society Incorporated.

CONTRIBUTIONS

All material for publication should be sent to:
Mr David Wilson
School of Computing Sciences
University of Technology, Sydney
PO Box 123, Broadway NSW 2007, Australia

Prospective authors may wish to consult manuscript preparation guidelines published in the February 1991 issue. The paragraphs below briefly summarise the essential details.

Types of Material: Four regular categories of material are published: Papers, Short Communications, Letters to the Editor and Book Reviews.

Generally speaking, a paper will discuss significant new results of computing research and development, or provide a comprehensive summary of existing computing knowledge with the aim of broadening the outlook of Journal readers, or describe important computing experience or insight.

Short Communications are concise discussions of computing research or application. A letter to the Editor will briefly comment on material previously appearing in the Journal, or discuss a computing topic of current interest. The term ‘computing’ is interpreted broadly to include descriptions of computer hardware and software, and major computer applications such as information systems, expert systems, computer networks and office automation.

Refereeing: Papers and Short Communications are accepted if recommended by anonymous referees. Letters are published at the discretion of the Editor, and Book Reviews are written at the Editor’s invitation upon receipt of review copies of published books. All accepted contributions may be subject to minor modifications to ensure uniformity of style. Referees may suggest major revisions to be performed by the author.

Proofs and Reprints: Page proofs of Papers and Short Communications are sent to the authors for correction prior to publication. Fifty copies of reprints will be supplied to authors without charge. Reprints of individual papers may be purchased from Associated Business Publications, PO Box 440, Broadway, NSW 2007. Microfilm reprints are available from University Microfilms International, Ann Arbor/London.

Format: Papers, Short Communications and Book Reviews should be typed in double spacing on A4 size paper, with 2.5 cm margins on all four sides. The original, plus two clear bond paper copies, should be submitted. References should be cited in standard journal form, and generally diagrams should be ink drawn on tracing paper or board with stencil or Letraset lettering.

Ideally, a hard copy, a 3½” Apple disk in QuarkXpress or Pagemaker, alternatively Microsoft Word is required. Straight text papers may be submitted in ASCII on 3½ disk.

Papers and Short Communications should have a brief Abstract, Key word list and CR categories on the leading page, with authors’ affiliation as a footnote. The authors of an accepted paper will be asked to supply a brief biographical note for publication with the paper.

Copyright © 1994 Australian Computer Society Inc.

Production Management:
Associated Business Publications Pty Ltd
(ACN 002 328 154), Room 104, 3 Small Street, Ultimo, NSW 2007 (PO Box 440, Broadway, NSW 2007), Telephone (02) 212 2780, 212 3780, Fax (02) 281 4594.

All advertising enquiries should be referred to the above address.

Publication Board
J.M. Hughes, Director

Editor
D.N. Wilson, University of Technology, Sydney

Associate Editors
B. Howarth, University of Technology, Sydney
G.C. Low, University of New South Wales
A. Moffat, University of Melbourne
M.E. Orловская, University of Queensland
K. Raymond, University of Queensland
E. Sonenberg, University of Melbourne

Editorial Committee

Office Bearers
G. Trinder, President
J. Manders, Vice-President
R.W. Tisdall, Vice-President
G.W. Dober, Immediate Past President
R.G. Heinrich, National Treasurer
J.E. Fowell A.M., Chief Executive
P.O. Box 319, Darlinghurst, N.S.W. 2010
Telephone (02) 211 5855
Facsimile (02) 281 1208

SUBSCRIPTIONS

The annual subscription is $50.00 local, $65.00 overseas. All subscriptions to the Journal are payable in advance and should be sent (in Australian currency) to the Australian Computer Society Inc.

Membership: The current issue of the Journal is entitled to receive one copy of each issue of the Journal published earlier in that calendar year. To ensure receipt of all issues, members should advise the Branch Honorary Secretary concerned, or the National Secretariat, promptly, of any change of address.

Members: Membership of the Society is via a Branch. Branches are autonomous in local matters, and may charge different membership subscriptions. Information may be obtained from the following Branch Honorary Secretaries:

ACT: GPO Box 446, Canberra 2601. NSW:
Suite 602a, 149 Castlereagh Street Sydney
2000. Qld: PO Box 135, Ashgrove 4034. SA:
GPO Box 2423, Adelaide 5001. WA: GPO Box F520,
Perth 6001. Vic: Locked Bag 36, South Mel­
bourne 3205. Tas: PO Box 216, Sandy Bay
7005. NT: GPO Box 213, Darwin 0801.

Printed by MAPS Litho Pty Ltd, Unit 3, 8 Aquat­
ic Drive, Frenchs Forest, NSW 2086.
This Special Issue on Software Engineering consists of a selection of papers from the 1993 Australian Software Engineering Conference held in Sydney from 27th September to 1st October 1993.

In its broadest sense, Software Engineering intends to convert software development into an Engineering discipline in which systems can be built with predictable properties, using known technology and to pre-determined cost/time schedules. The Australian Software Engineering Conference, a joint effort by the Institute of Engineers, the ACS and the Institution of Radio and Electronic Engineers Society, is the national forum for the presentation of new developments and research produced by a rapidly growing industrial and academic community. Work such as this is of major significance to the IT profession, and the ASWEC community is delighted to make the best of the refereed papers available to a wider audience through the Australian Computer Journal. Accordingly, preference has been given to those papers which, in addition to their academic rigour, addressed topics not normally receiving wide exposure.

The consequences of the evolution of project management methods are explored by Connaughton and Dampney (Surviving Paradigm Shifts in Software Development Technology: A Management Case Study of Industrial Experience). Their analysis explores two actual cases of paradigm shifts, one from a traditional to an object oriented methodology, providing practical advice for managers. Jayaputera and Cheng’s paper (SoftEAM: A Design History and Justification Maintenance Tool) deals with mechanisms and tools for recording the design process. Their tools, designed to support the CCITT System Description Language, allow the justification for design steps to be captured and to be replayed. Veschoor and Low’s survey of re-usability (Software Reusability in Australia) provides unique data on the extent and nature of the Australian IT sector’s state of practice. The paper suggests that reuse is recognised and practised to a greater extent than previously thought. Methods of displaying the structure of Ada programs allowing structural visualisation and analysis for maintenance and evaluation are presented by Slade and Parker (SEE-Ada - Software Evaluation Environment for Ada). A number of different architectural views are provided. Inter-working with the AdaMAT/D tools provides a basis for analysing sub-systems with questionable metrics. Bailes and his co-workers provide an overview of the techniques required for re-engineering software (Generic Re-Engineering Environment Design Criteria: An Evaluation of the Software Refinery™). Finally, Hoffman and Strooper address the issue of testing C++ classes (Graph-based Class Testing), using a combination of test oracles and testgraphs.

The Editors wish to thank the authors and reviewers, and the IREE for permission to reprint these papers, and, of course, the ACS for making the ACJ available for this purpose.

Karl Reed
Amdahl Australia Intelligent Tools Program
La Trobe University

Ross Jeffery
Information Systems
University of New South Wales
Surviving Paradigm Shifts in Software Development Technology: A Management Case Study of Industrial Experience

Colin Connaughton
CSC Australia Pty Ltd,
GPO Box 900, Crows Nest,
NSW 2065 Australia
Email: Colinc@Sydney.dialix.02.au

CNG (Kit) Dampney
Associate Professor,
Department of Computing,
Macquarie University, NSW 2109 Australia
Email: cdampney@mpce.mq.edu.au

This case study of industrial experience concerns the evolution of project management methods to cope with changes in development methods. In the projects studied the technical problems which emerged were, in many respects, the most tractable of the issues encountered. Adopting new development methods certainly introduced additional problems, but it was managing the change in project management needed that was the real challenge. As it turned out, the changes required for the development approach were sufficient to require shifting the mind set—which we can now recognise was a paradigm shift. That is, the established development methodology was replaced by a new approach requiring new ways of thinking. The essential result from the case studies, and insights from other fields, is that project management concerns in the development process are distinct and must be separated from the technical issues of development method.

Keywords: Paradigm shift, development methodology, development process, case study. CR categories: D.2.9, K.6.1

Copyright© 1994, Australian Computer Society Inc. General permission to republish, but not for profit, all or part of this material is granted, provided that the ACJ’s copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Australian Computer Society Inc.

Manuscript received: October 1993.

1 PARADIGM SHIFTS — MANAGING CHANGE AT THE LEADING EDGE
The concept of “paradigm shift” is that the development of scientific knowledge is governed by commonly accepted principles and laws which periodically undergo major re-evaluation and structural change, as a consequence of accumulation of knowledge as described in Kuhn (1970).

Essentially a paradigm provides the framework of scientific laws under which phenomena are investigated and interpreted. Periodically, pressure develops on some point in the perspective, in the form of anomalous or unexplained events in the real world (as distinct from the world of the paradigm). Such pressure indicates an inadequacy in the paradigm, and eventually causes the old perspective to be abandoned and another one adopted. Thus, a paradigm shift occurs, and the consequence is significant disruption.

Such discontinuous shifts are, of course, common in the human experience. Learning, financial markets, history and government are marked by upheaval as much as we may wish them to be smooth and continuous. Major change would seem to be unavoidably disruptive, even more so for the people involved if the change is not recognised until it is too late. The key to surviving a paradigm shift is to recognise when the limits under which “normal” practice operates have been exceeded, and a new approach is needed.

The purpose of this paper is to examine the impact of paradigm shifts in software development methodology on software development project management practices. In particular, we focus on managing the changes to project management that such shifts cause. In the arguments presented in this paper the advancement of scientific knowledge includes the advancement of management practices. As natural science models the physical world, so, in order to monitor and control a project, management practice presumes a model of the activities of technologists. Pressure on management practices results from failure to model activities sufficiently accurately, or from the need to model new activities resulting from evolution of the technology.

1.1 Change in a stable technological environment
It is challenging enough to manage a project under the assumption that paradigm shifts are not occurring in the technology. As a first step it is essential to recognise and manage risk. So far as risk caused by difficulties with the technology is concerned, the maturity of the development processes being managed is a key indicator. Maturity implies stability, lack of change, and therefore lower risks. Hence the efforts by some major learned and professional groups such as the Software Engineering Institute at Carnegie Mellon University (Humphrey, 1990) to measure software development maturity.

It is well known, e.g. Cash et al (1988), that project risk is increased by the differences in project size, technology, and complexity relative to the knowledge and experience of those involved. Furthermore, risk is enormously increased if the
required outcome is not well understood, as occurs in all too many projects. Such risk is caused by management difficulties in effectively communicating with the customer, which in turn may be caused by lack of shared experiences. In summary, risk is indicated by the level of pioneering and experience within the environment of the project and customer groups involved.

Where projects are pressing the current state of the art or where the customer is unable, or insufficiently advised, to comprehend the state of the art, then that risk becomes unavoidable. In either case the technology being used is relatively advanced compared to the project's environment.

Even so, such risk, if recognised, can be contained, despite the many case studies of failed projects caused by managers not controlling risk. Unfortunately such project failures continue and the profession of software engineering needs to address the problem. But that is not the purpose of this paper. The issue at hand is managing when the development process is changed by the progress of technology.

The management of development process change, described by Humphrey (1987), envisaged and reported a controlled, predictable evolution, quite unlike the collapse under pressure of Kuhn's paradigm shift. The basis of Humphrey's change management is stated to be that

"...the process itself must first be defined and documented. The key measurements are then established, data gathered and analysed, and corrective actions implemented".

This picture is a series of small changes leading to continuous process improvement, as proposed by the total quality management schools.

In stark contrast the disruptive nature of paradigm shifts suggests why the experience of so many organisations is so painful.

1.2 The impact of paradigm shifts on software engineering

The indications are that important projects have failed because major shifts are occurring in the paradigms governing software engineering. More to the point the evidence from the way that science and technology progresses suggests that we can expect a relative high frequency for the impact of paradigm shifts on software engineering compared to the older traditional disciplines of civil engineering and even electrical engineering.

The science behind software engineering is immature (Shaw, 1990) and still shallow relative to other science. Hence the probability of being near the leading edge of scientific knowledge is greater. The leading edge is, by definition, exposed to major shifts. For example, even physics which is a very mature science is being subject to major paradigm shifts in new areas such as astrophysics, fundamental particle physics, chaotic phenomena that impact biological processes and so on. Yet, civil engineering and mechanical engineering projects that apply classical physics can depend on a very stable underlying paradigm.

The question therefore is — how can the impact of paradigm shifts be recognised and thereby be managed?

Nowhere is the impact of paradigm shifts more evident than in software systems development, where established methods are widely thought to be inadequate. The "software crisis" is known to customer and developer alike to denote the history of late, over budget projects which delivered less than useful systems. Customer demands for improvements in productivity, speed of development and system complexity place further pressure on current development methods, and encourage early (and risky) adoption of new tools and approaches.

Since the basis for development project management (the "software development life cycle model") is largely empirical, paradigm shifts are particularly painful. Projects working at the limits of a paradigm, whether due to size, performance or sophistication, behave in ways which are unexpected and rarely favourable. Once a paradigm shift is made, the situation becomes worse until a body of experience with the new technique is established. Little or no guidance is available to the manager who must manage such a transition, since life cycle models (with, perhaps, the exception of Boehm's Spiral model (Boehm, 1988)) deal with steady state practices.

Quite clearly, paradigm shifts impact the methods of software systems development. There continues to be a flood of new concepts and practices being proposed for software system development methodologies in various areas of specialisation including information systems, computational systems, and embedded control systems.

If it were only a question of managing the introduction of new methods, then surviving a paradigm shift would be challenging, but not so deep a challenge as the evidence from practice suggests that it is.

1.3 Multiple levels of software engineering processes

Understanding the impact of a paradigm shift requires a broader approach which seems best illustrated by a case example (see next page) that would be familiar to many people.

This case demonstrates that design and production are quite different processes. In software engineering terms the processes are called "systems development" and "use". The model in Figure 1 takes this one stage further by including the process of designing (and developing) the tools that are used to support the systems development process. Thus from a management perspective, software engineering is more than just a sequence of processes, it is several processes that must be managed separately, where each process has a distinct context and set of management objectives.

From the perspective of the person who will use the system in production or to provide a service, it is the system that is delivered that is important. The methods

---

1 The word "customer" is used instead of "user" in keeping with the current quality management drive to recognise that it is the customer that must be satisfied.
used for developing the system are a separate issue. Perhaps from a quality perspective they are important in the sense that the delivered system must have quality that will be sustained. However, delving into deeper development issues runs the risk of becoming distracted from the objectives of using the system.

1.4 Two views of a paradigm — the Product Focus and the Activity Focus.

Within the development manager’s perspective the paradigm governing the development process can be viewed from above and from below, as illustrated in Figure 1, by:

1. **What is to be done** — the development process must deliver a system into the **application domain** with the required capabilities for use, and

2. **How it is to be done** — the activities of the development process must be supported by the right tools and methods drawn from the **enabling tool design domain**.

This separation clearly distinguishes between enabling and application. Indeed in Figure 1, if the customers of the services were added above the top layer and the provider of R&D support technologies was added at the bottom, the distinction becomes evident at all three levels.

Manley (1990) distinguishes these two views, what and how, by separating the development process with his dual life-cycle model into the management life cycle and the engineering life cycle for a project, together with their interactions within an enterprise-wide context. This caused him

---

**Case example: A modern computer system production line**

A visit to a modern assembly line in a computer manufacturing company was very revealing. Sun Microsystems, a 3 billion dollar a year enterprise, manufactures workstation computers in their San Francisco factory. The assembly line itself is described as a low technology operation, but many of the individual production processes are high technology. The result is a production line that is both versatile and well designed.

As one watches, a box is moved along the line every minute or so. Each box holds a computer worth several thousand or tens of thousands of dollars. The pace is relaxed and the workforce mainly directs their attention to ensuring that quality is maintained and that testing is thorough. It is staggering to realise that this apparently simple assembly line generates $2 billion dollars worth of product a year!

Where is the value generated? The answer lies in the product design and engineering laboratories that set up the production capability. These laboratories together with closely managed co-ordination between production and design is where the value is generated. Permission to visit the product design and engineering laboratories required much higher authority.

---

116 THE AUSTRALIAN COMPUTER JOURNAL, VOL. 26, No 4, NOVEMBER 1994
"to gain a deeper appreciation for the incredibly complex web of human relationships and communication paths that affect large-scale software development projects".

The two views of a process governed by a paradigm are also suggested by Brooks' (1987) survey of the state of software development practice. Dealing, as ever, with pragmatic issues, he notes "The familiar software project, at least as seen by the non-technical manager, ... is capable of becoming a monster of missed schedules, blown budgets, and flawed products" (our italics).

The importance of Brooks' passing salute to "the non-technical manager" is that there are two views of a paradigm — that seen by the developer, and that visible to management. The two are at different levels, in much the same way as the separate levels of process suggested by our multi-layered process model. However, in another sense, the developer's paradigm is enclosed within that of the manager.

The two views correspond to the development process model and the development method. Boehm [6] summarises the distinction most clearly:—

"The primary functions of a software process model are to determine the order of the stages involved in software development and evolution, and to establish the transition criteria for progressing from one stage to the next. These include completion criteria for the current stage plus choice criteria and entrance criteria for the next stage. Thus, a development process model addresses the following software project questions:

1 What shall we do next?
2 How long shall we continue to do it?"

The distinction as Boehm describes it, is between a management model, which is product focused, and a development method, which is activity focused.

The importance of the product/activity distinction is also confirmed by management theory. Beer (1979) makes essentially the same point in discussing the mechanisms of management. His First Regulatory Aphorism states "It is not necessary to enter the black box to understand the nature of the function it performs". Managers who disregard this aphorism are characterised as — "They are prone to enter the box, hunting down 'causes' with the jawbone of an ass". Though the cause may be found it does not help in managing the current activity.

1.5 Paradigm shifts originate from both the enabling and application domains

There is a second insight provided by the multi-layered software engineering process of Figure 1. Paradigm shifts can originate from above or below the development level. Not only are there changes in the principles governing new technologies and methods, but there are changes in the principles governing their application. For example new scientific principles have lead to the development of communications technologies, but it is changes in the principles governing society's behaviour that are governing globalisation of communication services. So paradigm shifts may originate from either the application or the enabling domains.

In summary, paradigm shifts impact the development project manager from both directions, and each shift needs to be considered from both points of view.

1.6 Controlling the impact of paradigm shifts on project management practice

The key to effective project management is clearly defined inputs and outputs for the development activities. Paradigm shifts from the enabling and the application domain respectively cause changes to the inputs needed by a development activity or the outputs required from a development activity.

If the development group can agree on the characteristics of these inputs and outputs, then the activity is manageable as part of the development process. The paradigm shift can then be attempted with some chance of emulating Humphrey's orderly process improvement process.

However, this may be very difficult to achieve. Without experience of the technique in realistic scale use, the attributes of an acceptable intermediate stage product in the development process will typically be the subject of considerable argument. It may be necessary, at least initially, to manage the activity as research and development; the difference being, largely, that the form, and indeed the occurrence, of its outputs cannot be guaranteed.

Management of such research and development is inherently uncertain, but it can at least be de-coupled from the application system production activities, improving their chance of success. If the research and development activity yields positive results, its experience becomes the basis for defining product standards in the production project. Either the attributes of working products of the research and development activities form the first approximation to production standards, or the experience of the new method provides a tested basis for determining what the standards should be.

Thus painfully, but in keeping with the process of developing scientific knowledge, the software development process evolves by experimentation and systematic trial and test. Each advance in the enabling technology domain or in the applications domain leads inevitably to improvements in the development process. Unfortunately the inevitability cannot be scheduled or costed. There have been and will be many failures as the industry improves its development methodology. This should come as no surprise — methodology like technology, is the product of R&D processes that depend on the capricious outcomes of scientific discovery.

Confusing managing a methodology with managing deliverables simply transfers without good reason the risks of R&D to the project management process.

Recent introductions of distributed-system, object-oriented, and window technologies have had an impact as significant as
the changes from batch to on-line processing. Each of these shifts have raised the computing platform to new levels of functionality.

However, these methods have been accommodated, though with increasing difficulty, within the dominant stagewise and waterfall variants of the sequential development process model (see, for example, Boehm, 1988). This reluctance to change the management paradigm persists despite difficulties with the waterfall approach.

A major current pressure on the sequential development process model paradigm is the need to modify requirements, particularly human-computer interaction, as a development reveals improvements that will better satisfy the customer. Such project change activities are usually handled in an ad hoc manner until “normal” activities can be resumed. As a result, project change is unpredictable and high risk. A project change control process is typically implemented to address this.

In most projects, a point is reached beyond which change requests are refused because of such risks, and delayed until the system (which is known to be incorrect!) has passed acceptance testing. Some willingness on the part of developer and customer to acknowledge waterfall limitations and manage the risks of trialling alternative approaches is needed. This proves culturally difficult, since a paradigm shift in development process management, perhaps correctly, is seen to be much more risky than the paradigm shifts that govern methods evolution.

1.7 Cultural Inhibitors to Orderly Changes in Project Management Practice.

Manley’s (1990) new software development life cycle model, referred to in Section 1.4, explicitly identifies the separate management and engineering processes and communications paths between them. As Osterweil (1992) demonstrates, software process research only now provides the framework to show that software development process improvement is analogous to, and at least as difficult as, software development itself. Why is it that the software development community give these issues so little attention?

Furthermore, the distinction between product and activity focus provides a clean analytical split between the concerns of technical and management people, whilst clearly displaying the relationships. Why, then, is the distinction not more widely understood?

We believe that the answers lie in the culture of the software engineering profession. For example, in software systems development the manager who has not had significant technical experience is not respected. There is some justification for this, in that the issues that must be managed (and separated) are inevitably first evident as a mixture of technical and management problems.

On the other hand, the technical background of the majority of project development managers means that they may not bother to separate technical and non-technical aspects of the issues. This renders the problems inaccessible to those without technical grounding, and the situation is perpetuated. Worse, the importance of the separation between technical and management issues goes unnoticed:—

— they are not distinguished by the “technical” manager;
— the technical view is not understood by the “non-technical” manager;
— the management view is not even perceived by the “technicians”;
— nobody communicates between the groups.

Viewing systems development difficulties from both product and activity viewpoints emphasises the mixed management and technical nature of the problems. It is clear that in many cases, the problem can be separated into two sets of problems of different types. Typically, one set encloses the other in some sense, although sometimes they may be quite independent.

An example might be difficulties with development environment throughput, where a technical problem (CASE tool performance) is enclosed within a resource management issue (limited funds). The alternatives of modifying the tools, buying more hardware, or staff working shifts would need to be considered. The “missing” management issue is why there is no contingency budget available. An early recognition that adoption of new tools involved a “paradigm shift” risk might have led to some resources being set aside.

The consequences of all these cultural inhibitors are far-reaching. Technicians will often be “promoted” to management positions with little formal management knowledge or training. Managers with such training may not be considered for positions, because they lack the technical background. Software development management is thus cut off from a body of applicable experience and techniques because it sees itself dealing with problems that are peculiar to software.

Hence, when a paradigm shift occurs from the technical direction of the enabling domain, mechanisms for managing the development process changes that are needed may well be overlooked. At best it will be recognised that the transition to new methods has to be managed.

This is despite the prevailing culture of systems developers that management and technical aspects are intertwined. Even when the techniques required for systems development change due to improved technology, deliverables marking progress are still required according to the old development management regime.

What is clear in a paradigm shift is that not only do new technical problems arise, but the relative importance of deliverables changes as well. The key to management is to distinguish between the two and manage accordingly.

Amongst this background of interrelated paradigm views and a closed culture of technical management gnostics, there are ample opportunities for projects to go astray. The case study examines aspects of two software system development projects in the light of these influences.
2 CASE STUDY OF TWO PROJECTS IMPACTED BY A PARADIGM SHIFT

Before describing the projects, it should be noted that, from the case study viewpoint, the interesting parts of the projects are the things not done or done badly. Hence the picture painted is exaggerated in terms of pragmatic progress and success. No attempt is made to provide a balanced evaluation of the projects in overall terms as the focus is instead on evaluating and learning from their detailed experiences.

The projects studied are of different types, and come from two substantially different application domains. One is a pilot project undertaken to explore and demonstrate object-oriented techniques in management information systems (MIS) development. The other is a “production” project developing a large, embedded software system for military application.

In both cases, it was well recognised that there were significant changes in the technology being applied to the development approach. As a consequence management attention became focused on managing the technology transitions of the development methods and therefore overlooked managing changes to the development process itself.

2.1 Impact on development method

In the MIS project the object-oriented technique of amalgamating data and function, which was the subject of the investigation, was well understood at the conceptual level in terms of data models and business rules. Established practices addressed the need to amalgamate data and function, but in an ad hoc manner.

The embedded system project adopted automated tools because for the size of job it was infeasible without them. The expected role of the tools was to support a structured analysis approach which was well established and understood. However, the combination of the tools and problem forced adoption of unfamiliar modelling approaches (state transition diagrams and entity relationship diagrams). Overlayed on this was the need to adopt an object-oriented approach in order to address problems encountered on previous Ada language implementations. The object-oriented approach was consistent with established practice — it focused on aspects of analysis and design which were agreed to be important, and which looked as if they would alleviate earlier problems. Nonetheless, there was no direct experience of object-oriented development available to the project.

2.2 Impact on Managing the Development Processes

Both projects undertook management of their paradigm shifts caused by technology changes within traditional waterfall development process models.

The MIS project’s groups were managed informally. For example, there was no separate quality assurance function and no management process or external documentation to control standards. Development stages were “collapsed” as necessary, and detailed management reporting was required on an exception basis only.

In contrast, the embedded systems project activities proceeded under guidance of military standard requirements for process and product characteristics. Formal project management, separate quality assurance and copious working documentation are all characteristic of this culture. The activities relating to methods development were not separately managed, since they were essentially similar to infrastructure development activities (interpretation of standards, development of project specific procedures and guidelines, for example) routinely undertaken on such projects.

In summary, little attention was given in either project to managing changes in the development process itself. Development process management was far more influenced by their existing informal and formal approaches respectively.

3 CASE ANALYSIS

The two projects pose several questions and demonstrate various consequences concerning the management practices employed.

3.1 Where did the method go?

The urgent nature of shortcomings in software development methods leads to considerable pressure for adoption of new approaches, whether the goal is improved productivity, product quality, or time to develop. This leads development projects to adopt new methods for which there is little prior experience. As a result, the projects must evolve and document, if not invent, practices as work proceeds. Revision of practices and retraining of staff are inevitable.

If this is not done, and schedule and budget pressures usually do work against it being done, then the substance of the new method itself may become the subject of argument. Various informal “dialects” of the method will evolve, reflecting individual experience and preference. Many of the variations will be of little significance to the essential method, but this can be difficult to demonstrate. The sorts of argument that have attended the evolution of individual programming languages, or of language editors, are good examples of the zeal with which such variation is defended.

Moreover the variations obscure the status of activities, and the possible need for correction to the method. Judging the impact of minor variations on a well established method is difficult enough. When the method is unfamiliar, the difficulties are compounded. When the project is complete, there may then be little chance of repeating the approach, if it was successful, or evolving it, if it was not. At least one paradigm shift will have occurred, but their substance and purpose remains unclear.

In the MIS project, participants in the pilot activity were given general training in the Eiffel language, and hence in object-oriented development. Subsequently, their work methods developed individually and informally, as an extension of the information engineering practices already prevailing. No

3 Like other creative activity, individual ego and the desire to make one’s personal stamp is often associated with R&D.
formal record of issues, queries and assumptions was kept, nor was any study of differences in individual approach made.

Training material for the development staff was subsequently produced by the senior member of the pilot group, and reflected a view of how object-oriented development ought to be done, rather than how it was done in the pilot activity. Some extrapolation of practice from the pilot activity to a much larger development group was necessary, but the "technology transfer" from the pilot group was never precisely identified.

The development group then proceeded to develop their own personal styles of object-oriented approach, since their training had been much more advisory than prescriptive. Some time later, a document was produced which specified "the method", but again, this reflected a view of how things ought to be done, rather than how they were done. No formal study or review of individual practices was undertaken.

The status of the object-oriented approach was thus unclear, although nearly all the staff involved seemed to have confidence in the approach, and their ability to work within it! It may well have formed such a consistent extension of their existing data modelling approach that no serious difficulties were encountered. Alternatively, they may just have proceeded as they had always done, simply using Eiffel as a specification notation.

Whatever the true situation, there is not sufficient existing documentation to clearly describe what was done. As the key staff involved in the activity are now dispersed, it would entail significant effort and risk to repeat or evolve the method developed on the project.

A similar situation occurred in the embedded system project with the use of CASE tools. Initial training given to analysis and design staff did not include any guidelines for the style of diagrams and data dictionary entries. Previous experience with structured analysis and design methods had not encountered any difficulties with consistent use of symbols and data dictionary entry formats.

However, these earlier projects had involved much smaller analysis/design groups, working in close contact. By comparison, the embedded system project involved two quite large analysis groups, working more-or-less independently. Initial indications of diverging approaches were apparent from the technical queries raised with the CASE tool supplier. By the time early reviews of analysis models were undertaken, it was apparent that different, and in some cases inconsistent, local styles had developed. The semantics associated with data flow diagrams in particular, where published methods leave a good deal unspecified, raised a number of issues.

The solution in this case was to establish a working group to agree on standards for the model representations. However, some level of rework of the existing models was required for both groups, and some duplicated effort had already produced different solutions to common problems. Whilst not a serious difficulty, the experience demonstrates the value of standards development in large groups. This is particularly so where there is not a body of prior experience to guide experimentation, and communications are limited by organisation structure, division of responsibility or geography.

3.2 Is "good enough" good enough?
When making a paradigm shift, it is important to maintain clear sight of the goals that have been adopted. On a pilot, or "technology demonstrator", project, it's not necessary to produce a working product, just one that clearly could work. Conversely, on a production project trial, it is essential to produce a working product, even if this means deferring some questions relating to the development methods.

For the MIS, trialling the new paradigm in a production development environment required careful management to ensure the needs of exploring the methods, and those of delivering a working system, were reasonably balanced.

The aim was not to produce a set of development methods, tools and training procedures which could be generally adopted in the organisation. Rather, it sought to improve the pilot activity techniques with minimum possible investment, consistent with successful delivery of the operational system. On completion of the project, the place of the object-oriented techniques in the range of main stream development approaches could then be reassessed, and further research and development activities considered.

Inevitably, strong pressure developed to produce production quality tools. In some cases, extensions were necessary to fix problems that would otherwise have jeopardised the application system. In other cases, clear improvements in the productivity of the application developers were the justification. No overall commitment was made, but case by case and feature by feature, the sophistication of the support software grew.

Eventually, the support software had become middle-aged and overweight in comparison to the antecedents used in the pilot activity. Creeping featurism was probably a factor, as was the tendency to try and address the general case of the problems encountered, rather than seeking a more limited solution. However, simple, optimistic underestimation of the difference between a test-bed and a production environment was the real problem.

In such situations, it may well be that a project cannot be completed under the new paradigm without major changes to resource budgets and schedules. Alternatively, a reversion to more traditional methods may be required to meet project commitments. Whichever course is adopted, a clear focus on the project deliverables or the methodology goals is needed, or both may be missed.

3.3 When is "enough" enough?
Whether the project is a pilot or a production trial, it is necessary to be prepared to terminate the activities if goals are not going to be met. The uncertainties of the paradigm shift make it essential that some alternative plan for the project is available, especially where others are depending on the project.
The situation is essentially that described by Brooks (1975) — when the souffle will not be ready on time, you have the alternatives of turning up the heat and burning it, or eating it half cooked. However, it is also possible to make a cheese sandwich instead; it is probably disappointing, but better than going hungry. The termination process, if necessary, should endeavour to preserve as much product, tools and experience as possible, against the probable need to try the paradigm shift again.

In the MIS project, concerns about the probable system performance, and the maintenance effort for the increasingly complex run time and utility software forced reassessment of the project. To this stage, a significant collection of tools, run-time software and underlying (“system”) classes had been developed, and experience gained had addressed most of the issues identified following the pilot activities.

However, experience of the method in the development of business functions was limited, since development activities built on the system classes depended on the availability of tools. The project was thus close to a turning point in terms of the return on resources applied. For some time, further effort would yield improvement primarily in the tools and run-time software, rather than further experience of the method. However, without improved application performance, there was little value to further application development.

The key factors of performance and support software complexity had changed. They now had to be judged against the original goal of improved object-oriented development methods. Specifically, the goal was to prove that object-oriented techniques applied to business system development would have minimum cost and risk impact to the application system development process.

The performance improvement required seemed achievable, but increased the complexity and scope of the support software. There was a risk that such extended function would bring with it new problems, which previously had been handled by operating system software. Effort and schedule might thus become extended by a series of unexpected problems.

The complexity of the tools and run-time software was of concern in terms of the skill level required for long term maintenance. There was still significant effort and likely delay to bring the software to useable commercial standard. Again, schedule delays were the major issue.

Ultimately, the option of completing the project by conventional methods was chosen, to the disappointment of all involved. However, there is typically no “best” strategy that will optimise returns to all parties for effort spent. A compromise choice must be made, taking into account not only the technical factors, but also the situation prevailing within the development, maintenance, operations and user organisations.

3.4 Consequences of laissez faire management

In the embedded system project, two distinct “schools” of analysis and design developed. The first advocated elaborating the analysis model until a level of detail was reached from which program modules could be specified. The model would be a “logical” representation, unconstrained by implementation issues. At that point, the model would be “distorted” as necessary to produce an implementable system. This is essentially the approach of Ward and Mellor (1985). “Computer based” objects4 would be identified as groupings of data and function within the model.

The second school advocated producing an analysis model to the minimum level of detail necessary to unambiguously define requirements. Once this was achieved, a new logical model would be produced, starting with the identification of “reality” objects appropriate to modelling the application domain. This object-oriented model would be required to support all requirements present in the original (structured) model, but need not adopt any of its structure. Issues of computer based implementation would be largely resolved “bottom-up” during identification of the objects.

In the absence of direct experience with object-oriented development, it was not possible to choose between the alternatives. Even with the assistance of hindsight, consultants and published literature, it is not clear that either approach would always be preferred. Since the two schools had developed in project areas which would produce independent subsystems, there was no technical imperative to force adoption of a single approach.

Since one approach was “top down”, and the other essentially “bottom up”, the effort ratios for these activities were biased quite differently. The more structured, top down, approach would defer questions of implementation and object/package selection until late in the design phase. The more computer based object-oriented, bottom up, approach would deal with these issues quite early, and would truncate the analysis phase to allow this to occur in the design phase.

Hence management had to deal with two sets of issues arising in two different, unfamiliar approaches to systems development. It lost an opportunity to compare the progress and problems of two independent teams both following variants of an object-oriented approach. In the absence of prior experience with object-oriented techniques, this was a critical loss. At least some of the management uncertainty surrounding progress assessment, job size validation and productivity measurement could have been alleviated by comparison of activities in the parallel development groups.

4 OBSERVATIONS AND CONCLUSIONS

The essential result from the case studies is that project management concerns in the development process are distinct and must be separated from the technical issues of development method. How then should the development process respond to major change in method and vice versa?

4 The term “computer based object” is used to distinguish from system approaches that begin with “objects” or extended entities inferred from the reality of the application domain.
4.1 Who’s “right”?
It must be understood that most of the issues and questions identified here don’t have “right” or “wrong” answers in any absolute sense. Life cycle development models rest on empirical and judgemental guidelines for what should be done, how long it should take, and what the resulting product should look like. However, it is possible to depart substantially from a particular life cycle profile, and still get a good result. It is not easy or safe, but it is possible. Paradoxically, such departures sometimes produce dramatically successful results — the enthusiasm for rapid prototyping is based on such experience. Such dramatic successes are to be expected — every so often a team happens to discover a design or project management method which really does improve the development process.

4.2 Recognising the situation
From the management viewpoint, these situations present endless difficulties. A successful project which is following a new paradigm, may look identical to a disaster following an established paradigm. Where doubt as to the true situation exists, management of intermediate stage products is essential. The developers must be required to justify their product’s characteristics, to external reviewers if necessary.

If this cannot be done, then the activity must be recognised as research and development, not production, and appropriate business risk containment strategies put in place. The nature of research and development will always be such that outcomes are uncertain, but the unexpected nature of paradigm shift problems probably causes more difficulty than the actual consequences entailed. Typically, the actual problem can be solved, but there is insufficient schedule time and/or resource budget available. Recognising the likelihood of this situation during project planning is half the battle.

4.3 Value rigidity
The case study projects provide a number of examples of the management disruption caused, unexpectedly, by a shift in development method. However, although it was not possible to predict the problems that would arise, both projects understood that difficulties would occur. Nonetheless, neither project chose to modify its project management model to take account of the risks this implied. In both cases, it was accepted that ad hoc management actions would be necessary, and presumably sufficient.

The readiness of the software developer to explore and embrace new methods contrasts strongly with the reluctance of the project manager to deviate from the traditional waterfall model. Clearly, it is undesirable to vary both development method and development process model on a single project, and this would have been a factor in both case study environments.

However, the project manager’s resistance to change is more probably an example of the value rigidity identified by Cox (1990) as a limiting factor in the spread of new technology:

“...The software industrial revolution, like all revolutions, is as much cultural as technological. It involves not only tools but values: deeply held beliefs about the nature of software, our role in its construction and use, and ultimately our ideas of good versus bad. Revolutions happen so slowly — and often displace one group by another — because of value rigidity, the inability to relax the pursuit of an older good to gain a newer one”.

4.4 What’s next?
To some extent, “the software crisis” reflects both the immaturity of software development as a professional practice, and the astonishing, sustained rates of change in the underlying software and hardware technology. The good news is that the software engineering discipline is maturing; the bad news is that it is not happening fast enough.

To keep this in perspective, we must not forget how complex the systems are that we are required to build. In his 1987 assessment of the state of software engineering, Brooks (1987) comments:

“Software entities are more complex for their size than perhaps any other human construct .... The complexity of software is an essential property, not an accidental one. Hence, descriptions of a software entity that abstract away its complexity often abstracts away its essence”.

We must therefore expect that management techniques capable of recognising and containing risk, whether from paradigm shift or traditional causes, will continue to be a sorely felt need.

The growing maturity of software development will inevitably produce more standardisation and codification of practices. This, in turn, will support the product versus process focus that management of the technology requires. It would be a pity if in the process, we miss the opportunity to develop better management models, with more explicit flexibility and a better view of the respective roles of developer and manager, of method and model.

ACKNOWLEDGEMENTS
The support and co-operation of Computer Sciences of Australia in the development of this case study material is gratefully acknowledged. Particular thanks go to the technical and management staff, whose willingness to analyse and discuss their practical experience has contributed significantly to the process.

REFERENCES
SURVIVING PARADIGM SHIFTS IN SOFTWARE DEVELOPMENT TECHNOLOGY


BIOGRAPHICAL NOTES

Colin Connaughton is a senior member of staff of CSC Australia Pty Ltd. He has over 23 years experience developing large scale software systems for defence, engineering and business application areas. These systems involve hard real time control, communications and general business applications. He is particularly interested in the systems development process, requirements modelling and representation.

Colin is a member of various university advisory committees including the Joint CSIRO Macquarie University Research Centre for Advanced System Engineering, Software Verification Research Centre at the University of Queensland, Basser Department of Computer Science, University of Sydney, and the School of Computer Sciences at the University of Technology, Sydney.

He is a member of IEEE, ACM, the IREE Society and a fellow of the Institution of Engineers (Australia).

Kit Dampney is an Associate Professor in Computing at Macquarie University where he directs the post graduate, post experience program in Computing and Information Systems. He is President of the Sydney Chapter of the Data Administration Management Association and a member of IEEE, ACM and the ACS. He has published widely in international journals in both information systems management practices and analysis methods.

Kit has a broad base of experience in information analysis acquired since 1970 through consulting to major companies and research. His continuing aim is to discover and understand the fundamental principles of information systems, to apply them in practice, and to share that understanding with others.
1 INTRODUCTION

In general, the software design process does not progress in a single linear path. Rather it spreads out and deviates for several reasons such as bug fixing, adjustment due to changes in the requirements, hardware limitations, etc. Any of these issues forces a designer to 'step' backward to one of the previous design stages and to redesign from that stage. In a large design project, backtracking to one of the previous design stages without a proper mechanism is hard. And since proper backtracking is not supported the process of designing itself becomes harder to manage.

The situation is exacerbated by the increase in the amount and size of information such as design specifications and justifications involved during the software development life-cycle. Furthermore, their interrelationships need to be captured and recorded so that they can be played back whenever a designer wants to review all the information. These pieces of information and their relationships, however, keep on growing in size and complexity along with the process itself, which stops only when the design is done. This complexity makes maintaining this information a very complex and time consuming task, yet very vital to the end product. Failing to maintain the design information and their relationships causes the changes made in the design process to tend to be unprincipled, ad hoc and error prone.

Although this problem has been identified for some time (Luqi, 1990), remarkably little has been done to solve this problem. Luqi (1990) suggests this is due to the lack of a traceable representation model for the process of software evolution.

The formal representation model presented in this paper helps to address this problem, in that it can capture the evolution of a software design specification and justifications systematically and consistently. However, rather than developing a generic formal model, we focused our development on the ITU (formerly known as CCITT) SDL (Specification and Description Language) (CCITT, 1988; Saracco and Tilanus, 1987) based design specifications. However, this does not limit one to further extend it in order to use it in different formal description techniques such as LOTOS (Bolognesi and Brinksma, 1987). Furthermore, the relationships between an SDL design specification and other software information such as requirement specification, auxiliary documentation, testing and implementation are not taken into account. That is the research work is restricted to only maintaining the design specification evolution history and justifications involved during the design process.

2 DESIGN EVOLUTION MAINTENANCE

Maintaining the evolution of the design artifact such as design issues, decisions and specifications is very hard and tedious for human beings. An appropriate model to capture this evolving information is needed. This is due to the fact that the pieces of information that are evolving and interrelated form relationships that are too complex for human beings to re-
member. As stated by Luqi (1990), computer assistance is essential for an effective and reliable evolution of complex systems because their representations and evolution histories are too complex for unaided human understanding.

The complexity of the evolution history is caused by the fact that the development paths normally form hierarchical tree representations and that there are some interrelationships between software components underlying the application software. The hierarchical tree representation is caused by backtracking performed by the designers and normally occurs a large number of times. The interrelationships between software components are normally caused by the direct outgrowth of the high coupling between the software components.

To accommodate the demand and make changes to the software system at the design level, an agent needs to understand not only the relationships among the components of that system, but also all the previous decisions that have been made. In almost every case, the agent needs to inspect the contents and purpose of the modules in the system (Ramamoorthy, Usuda, Prakash and Tsai, 1990) as well as the decisions and justifications made. Without this action, modifications to the software system are almost impossible to be done and hence meet the specified requirement.

For large software systems that are developed over a long period of time, such information is usually not available to hand nor even captured systematically, which makes retrieving such information time consuming. As a result, the development of such a software system is slowed down and is error prone.

A telecommunication software system is one of the instances where the size of the system is normally large and complex. Also telecommunication software systems are normally revised over a long period of time to accommodate new hardware or functionality enhancements.

Maintaining multiple versions of multiple software modules and their relationships is very complex, yet is not a new problem. Several tools have been developed to address this problem, such as RCS/SCCS version management systems (Rochkind, 1975; Tichy, 1985) and configuration management systems.

Although these tools exist, none of them are perfectly suited to be used with telecommunication software design, in particular with the ITU Specification and Description Language. Our study has identified that RCS/SCCS and their families are not suitable to maintain the evolution history of SDL-based design specifications because they only provide a good support for keeping track of multiple revisions of a software module (it is assumed that a software module is stored in a single file), but fail to maintain and remember the relationships between a set of software modules of which a software system is composed. Furthermore, they can only be used on text-based objects. For an object that consists of graphical constructs/symbols such as ITU SDL, these systems cannot be used. Consider the two revisions of an SDL process described in Figure 1.

![Figure 1: Graphical Representation of Process Nothing.](image-url)

**Figure 1:** Graphical Representation of Process Nothing.

**Figure 2:** Textual Representation of Process Nothing.

The textual representation of the previous graphical representation (Figure 1) is illustrated in Figure 2. By examining the graphical representation (SDL/GR) of the two revisions of Process Nothing, one can immediately distinguish the difference between the two. In the first revision, there is a directed arc connecting constructs Input2 and Task1. In the second revision, however, this arc is replaced with the one that connects constructs Input2 and Task2.

However, if the textual representation of the two revisions is applied to RCS/SCCS (recall that RCS/SCCS only works on text-based objects, hence only the textual representation of the two revisions can be used) then the deltas derived by RCS/SCCS do not semantically make sense to the agent. Figure 3 illustrates the deltas generated by RCS/SCCS.

![Figure 3: The Graphical Illustration of The Deltas Produced by RCS/SCCS.](image-url)

**Figure 3:** The Graphical Illustration of The Deltas Produced by RCS/SCCS.

The textural representation of the previous graphical representation is shown in Figure 2. By examining the graphical representation (SDL/GR) of the two revisions of Process Nothing, one can immediately distinguish the difference between the two. In the first revision, there is a directed arc connecting constructs Input2 and Task1. In the second revision, however, this arc is replaced with the one that connects constructs Input2 and Task2.

However, if the textual representation of the two revisions is applied to RCS/SCCS (recall that RCS/SCCS only works on text-based objects, hence only the textual representation of the two revisions can be used) then the deltas derived by RCS/SCCS do not semantically make sense to the agent. Figure 3 illustrates the deltas generated by RCS/SCCS.
From Figure 3 the textual difference between revision 2 and revision 1 of the Process Nothing can be seen. The summary of those deltas are illustrated in Figure 4.

\[ \Delta_1 = \text{JOIN L 18P01}; \]
\[ \Delta_2 = \text{INPUT Input1; L 11P01}; \]
\[ \Delta_3 = \text{INPUT Input1; L 18P01}; \]
\[ \Delta_4 = \text{JOIN L 11P01}; \]

Figure 4: The Four Deltas Generated by RCS.

Unfortunately, the four deltas generated by RCS do not tell the agent that in fact only one single arc has been changed. In fact, these deltas even deliver a false message that Input1 has been added and removed from the revision. This false message is conveyed by \( \Delta_1 \) and \( \Delta_2 \).

What is needed is a system that can inform the agent that the structure has been changed. That is, the link that connects Input2 and Task1 no longer exists. Rather, the Input2 is not connected to Task2. As such, the agent will notice the change in the structure of this software module.

Configuration management systems are also not suitable to be used as an assistant tool during a design process. The aim of configuration management is to co-ordinate and control the evolution of a software system, and is not to assist the agent during a design process. 'Assisting' means helping designers during the design process, not just co-ordinating and controlling all the information produced by them.

RCS/SCCS does not capture the design decisions made during the design process in a meaningful way. The decisions, which are in textual forms, represent all design decisions made rather than precisely explain the related part of the design specification. The textual form of decision description makes it almost impossible to backtrack to a specific revision of a software object. To jump back to a specific revision where a decision has been made, a designer needs to give a maintenance system the exact string that represents that decision. This includes leading and trailing spaces and non-printable characters. The situation is made worse by the fact that a word can have one or more synonyms and they can be used interchangeably.

To develop a design history and decision maintenance system, the basic concept of what a software system and design state are needs to be defined. During a design process, software systems go through a series of stages until they reach a stage where an agent thinks that the design is complete. A design is complete if all the requirements are answered. These stages are referred to as design states. We define a design state as a stage at which a snapshot of a software component is taken. At this point our model is analogous to version management systems such RCS (Tichy, 1985), SCCS (Rochkind, 1975), etc. We define a software system as a set or collection of software objects. The software object definition provides the key to derive proper deltas. It defines the granularity from which the deltas are generated. We define a granularity as a primitive/basic construct that is made up of a language and is the smallest entity that is semantically understood by human beings. Choosing too coarse a granularity will cause the deltas generated not to provide meaningful feedback. On the other hand, choosing too fine a granularity causes the deltas generated to consist of all unnecessary information that an agent might not want to know. For instance, defining a software object as the modules which compose a software system causes the deltas generated to show the difference at module level. Likewise, defining a software object as a character in a design will cause the deltas generated to contain only the difference in terms of characters. This is considered as unnecessary information.

RCS/SCCS defines a software system as sequences of lines of characters. Thus a software object in these systems is a line of characters and its position in the system itself. However, our study shows that such definition is not appropriate when the position of software objects does not play an important role but their interrelationships do. Based on our understanding of the granularity notion, we identified that an SDL-based design specification consists of a set of SDL objects that can be of types: structure, atomic, interface or link. A structure object is an object where there is an underlying structure, i.e. an object that can be decomposed into a set of objects. Any of these underlying objects can be any of the four types previously illustrated. Block, Process, Procedure, Macro, Service constructs are examples of structure objects in the SDL environment. An atomic object is the finest granularity type of object. This type of object cannot be decomposed into other objects. In the SDL environment, Input, Output, Procedure Declaration constructs are considered as atomic objects. An interface object is considered as a special type of structure object. This type of object inherits some of the properties of a structure object. That is, an interface object can be decomposed into other types of objects. However, this object serves a special purpose in that it represents the connectivity between structure objects. Channel and Signal Route are examples of this type of object in the SDL environment. A link object is an object that represents the connectivity between atomic objects. This object identifies what is the 'source' and what is the 'sink'.

Using this object notion we have achieved a maintenance technique at object level as opposed to RCS/SCCS's physical/file level maintenance. We believe the file level maintenance technique is not appropriate in the current software engineering environment. For instance, if a file consists of N number of modules then their evolution history becomes lost because RCS/SCCS does not consider them as important objects, and treats the difference at file level. For instance, if module x in that file is changed, RCS/SCCS cannot report that in fact
module x has been changed as opposed to telling that the content of the file has been changed. With the object level maintenance technique such misleading information will not occur. In this case, the maintenance system will appropriately inform the users that object x has been changed. Such a notion will allow more than one software module to be placed in a file. In fact, in our model, designers are ‘encapsulated’ from the file notion. They are only concentrating on the evolution of objects they are creating without any concern of where and how those objects are stored. We believe that such a notion is very important.

In our model, deltas are generally defined as the difference between two revisions. However, the primary difference between deltas in the traditional version maintenance systems and ours is that our deltas are not only for storage space optimisation but also to be interpreted and used as one of the bare-bones to develop a better maintenance system. Since a software system is composed of a set of interrelated software objects then deltas are generated by ‘comparing’ these software objects. ‘Comparing’ means matching all the attributes contained in an object. We are not concerned with the speed of information retrieval; therefore, we follow the notion of reverse delta introduced by RCS. As noted earlier, we model a development process as a series of design states. However, the ancestral tree proposed by RCS/SCCS does not conform to this notion due to the delta numbering scheme used. We do not follow such a numbering scheme as it does not reflect the actual development path; that is, there is no guarantee that revision 1.4.1.2 was ‘checked-in’ earlier than revision 1.4.2.1 or vice-versa. With such a notion, backtracking to a previous design state is unfortunately quite hard since there is no information that tells which one is the previous state of a given design state. What we propose is to provide an ‘embedded’ code within a design state itself. This embedded code represents the occurrence of a design state; that is, when it is created. For instance, the occurrence code can be as simple as a positive integer or even a time stamp. By embedding the design state’s occurrence, chronological backtracking is handled efficiently. With an appropriate user interface, a designer can browse the design states sequentially. This ‘playing-back’ mechanism is an important support for a maintenance system, in that it lets a designer recapture his/her work; that is, what was done, decisions that have been made, etc.

The ancestral tree which represents a development path, is a directed tree which is composed of one main trunk and several branches. A main trunk is defined as a path where the main development of a software system occurs. A branch is defined as a path that deviates from a main trunk or another branch. Branching from the main trunk is normally due to experimentation, bug fixing, etc. Since changes in the branches sometimes need to be incorporated into the main development path, these changes need to be propagated. Deltas propagation is defined as a process of incorporating changes between two revisions. Deltas propagation is analogous to RCS’s ‘three ways’ merging technique (Tichy, 1985), that is merging two revisions with a common ancestor. However, rather than comparing and determining the segments of objects that are (a) the same in all three revisions, or (b) the same in two revisions, or (c) different in all three, we use the notion of ‘propagating’, where only two revisions are compared. The result of this operation is then broadcast to the third one. This kind of technique is less expensive if applied to a set based collection (recall that a software system is modelled as a set of software objects), yet will produce the same result as three-ways merging.

Deltas propagation uses a ‘two-ways comparison and propagating’ mechanism. Three ways comparison can be very expensive where the deltas are overlapped with each other. This is considered as an advantage of deltas propagation. However, one of the major drawbacks of the deltas propagation is that it only works well when the revisions are composed of a set of objects, but not a sequence.

The object classifications presented previously are used to represent an SDL-based specification. For instance, a Process is represented as a structure object while a Task is represented as an atomic object (note that for the sake of simplicity, other attributes of these objects are not mentioned).

It should be noted that the actual structure of the specification is preserved. This is because the objects of type link and interface actually capture this connectivity between objects of type structure and atomic. Therefore a set of these types of objects are actually a graph representation of an SDL specification. This graph is referred to as intermediate graph representation (IGR).

Figure 5 shows the relationship between an SDL specification and its intermediate graph representation.

Using the notion of the IGR the deltas between two revisions then can be generated and interpreted. First, the two revisions are converted into the equivalent IGRs. These IGRs are then ‘flattened’. Flattening a graph means that all the nodes and links that are associated with that graph are translated into a set of objects (i.e. nodes and links). The objects from two revisions are then compared and the differences are the deltas between the two revisions (Note that an object can have zero or more attributes ‘embedded’ in it besides its own unique identifier). Hence, the Process Nothing in Figure 1 can be represented as two IGRs as follows:

Figure 5: SDL Specification vs IGR.
By using set theory, the IGR representation above can be "flattened" as follows:

\[
\begin{align*}
R_1 &= \{ A, B, C, D, E, F, G, H \} \\
R_2 &= \{ (A,B), (B,C), (B,D), (B,E), (C,F), (D,F), (E,G), (F,H), (G,H) \}
\end{align*}
\]

By comparing these SDL objects, the delta between the first revision (R1, and R1) and second revision (R2, and R2) is obtained.

\[
\begin{align*}
\Delta_{R_2 - R_1} &= \emptyset \\
\Delta_{R_2 - R_1} &= -(D,F) \text{ and } + (D,G)
\end{align*}
\]

And hence

\[
\Delta_{R_2 - R_1} = -(D,F) \text{ and } + (D,G)
\]

The minus sign means that link (D,F) is removed from revision R2, while the plus sign means that link (D,G) is added to revision R2. Hence by looking at this information the designer knows that instead of object F, object G is now connected to object D. Also, they know that only the 'connectivity' has been changed since the first revision, because the deltas only consist of the link type objects.

With this IGR and deltas notion, we are able to maintain the evolution of SDL-based specifications at object level. However, the situation is more complex in that some of the objects in the IGR can be a structure typed object. This type of object can be further decomposed and has its own evolution history. Objects such as System, Block, Process, Procedure, Macro, Services are some of the examples. We identify that recursively triggered evolutions can be nicely represented by using the IGR and design state notion explained previously. An ancestral tree that represents an evolution history is 'attached' to objects of type structure or interface. Therefore, in a typical SDL specification the evolution can be illustrated as in Figure 7.

A typical SDL specification evolution starts with a set of design states at the System level. Since at this level, a number of Blocks or Channels (which are structure and interface objects themselves) are defined, then they have their own set of evolutions (their evolution histories are captured and represented by a set of design states as well). A similar explanation holds for Block level where a number of processes are defined. At process level, a number of atomic and structure objects can be defined as well. For instance, Macros, Task, Input, Output, Procedures, etc. can be defined at this level. For the structure type objects (such as Macros and Procedure), they can have their own design evolution history. Each design state is 'tagged' with a status indicator. The tag can be any of: current, alternative, rejected and experimental. Current status is to indicate that the design specification of this state is the one that is currently being used. Alternative status is to indicate that the design specification of this state is an alternative and can be used if necessary. Reject status is to indicate that the design specification of this state is wrong and rejected but is kept for historical purposes. Finally, Experiment status is to indicate that the design specification of this state is unknown and is still under experimentation.

3 DESIGN JUSTIFICATION MAINTENANCE

To capture and maintain design decisions, the notion of design process needs to be identified first. To date there is no rigorous definition of what a design process is. A design process is defined as an activity of identifying all issues which need to be solved and making decisions to solve them. In other words, the traditional notion of a design process as just an activity of writing design specifications or codes is extended. The reason for this is because writing a design specification is just a means of transforming an abstract logic of a design into a formal representation. A typical SDL specification evolution starts with a set of design states at the System level. Since at this level, a number of Blocks or Channels (which are structure and interface objects themselves) are defined, then they have their own set of evolutions (their evolution histories are captured and represented by a set of design states as well). A similar explanation holds for Block level where a number of processes are defined. At process level, a number of atomic and structure objects can be defined as well. For instance, Macros, Task, Input, Output, Procedures, etc. can be defined at this level. For the structure type objects (such as Macros and Procedure), they can have their own design evolution history. Each design state is 'tagged' with a status indicator. The tag can be any of: current, alternative, rejected and experimental. Current status is to indicate that the design specification of this state is the one that is currently being used. Alternative status is to indicate that the design specification of this state is an alternative and can be used if necessary. Reject status is to indicate that the design specification of this state is wrong and rejected but is kept for historical purposes. Finally, Experiment status is to indicate that the design specification of this state is unknown and is still under experimentation.

3 DESIGN JUSTIFICATION MAINTENANCE

To capture and maintain design decisions, the notion of design process needs to be identified first. To date there is no rigorous definition of what a design process is. A design process is defined as an activity of identifying all issues which need to be solved and making decisions to solve them. In other words, the traditional notion of a design process as just an activity of writing design specifications or codes is extended. The reason for this is because writing a design specification is just a means of transforming an abstract logic of a design into a formal representation. A typical SDL specification evolution starts with a set of design states at the System level. Since at this level, a number of Blocks or Channels (which are structure and interface objects themselves) are defined, then they have their own set of evolutions (their evolution histories are captured and represented by a set of design states as well). A similar explanation holds for Block level where a number of processes are defined. At process level, a number of atomic and structure objects can be defined as well. For instance, Macros, Task, Input, Output, Procedures, etc. can be defined at this level. For the structure type objects (such as Macros and Procedure), they can have their own design evolution history. Each design state is 'tagged' with a status indicator. The tag can be any of: current, alternative, rejected and experimental. Current status is to indicate that the design specification of this state is the one that is currently being used. Alternative status is to indicate that the design specification of this state is an alternative and can be used if necessary. Reject status is to indicate that the design specification of this state is wrong and rejected but is kept for historical purposes. Finally, Experiment status is to indicate that the design specification of this state is unknown and is still under experimentation.

3 DESIGN JUSTIFICATION MAINTENANCE

To capture and maintain design decisions, the notion of design process needs to be identified first. To date there is no rigorous definition of what a design process is. A design process is defined as an activity of identifying all issues which need to be solved and making decisions to solve them. In other words, the traditional notion of a design process as just an activity of writing design specifications or codes is extended. The reason for this is because writing a design specification is just a means of transforming an abstract logic of a design into a formal representation. A typical SDL specification evolution starts with a set of design states at the System level. Since at this level, a number of Blocks or Channels (which are structure and interface objects themselves) are defined, then they have their own set of evolutions (their evolution histories are captured and represented by a set of design states as well). A similar explanation holds for Block level where a number of processes are defined. At process level, a number of atomic and structure objects can be defined as well. For instance, Macros, Task, Input, Output, Procedures, etc. can be defined at this level. For the structure type objects (such as Macros and Procedure), they can have their own design evolution history. Each design state is 'tagged' with a status indicator. The tag can be any of: current, alternative, rejected and experimental. Current status is to indicate that the design specification of this state is the one that is currently being used. Alternative status is to indicate that the design specification of this state is an alternative and can be used if necessary. Reject status is to indicate that the design specification of this state is wrong and rejected but is kept for historical purposes. Finally, Experiment status is to indicate that the design specification of this state is unknown and is still under experimentation.
form of notation which can be interpreted by a human being. Furthermore, the entire development process can in fact be viewed as a continuum of decisions from choices about initial high-level system layout down to data representation and control structure (White, 1982). To adequately capture the impact of these design decisions in a design process, they need to be captured and recorded, and hence not only the design specification itself needs to be captured and recorded.

Design issues and decisions have to be captured in a systematic way for historical purposes (Mostow, 1985). Capturing and recording these elements in a more formal and systematic way will allow a designer to backtrack to any of the previous design states at which an issue is introduced or a decision is made. Capturing these elements of design justification should not be in textual form only, because they can be expressed and interpreted in many ways. Unfortunately, many of the design tools still use standard textual descriptions to capture design decisions. Capturing design decisions in such a manner, only provides a 'description' of design decisions. Some of the examples of such tools are: PSE (Programming Support Environment) (Tichy, 1982), ESE (Evolution Support Environment) (Ramamoorthy, Usuda, Prakash and Tsai, 1990), Gandalf (Habermann and Notkin, 1986) and Cactis (Hudson and King, 1988).

In the model presented here, design justification is composed of two elements — design issue and design decision. A design process starts with an action of collecting all the design issues that need to be solved. The action is then continued by making decisions on how to 'answer' those issues. In most of the cases, these decisions 'stimulate' other new issues to be answered. The process is repeated until all the issues are solved. Furthermore, in practice, it is common for a designer to make a number of decisions for a given issue. These 'decision alternatives' are analogous to the design choices discussed by Mostow (1985).

By identifying all the issues and subsequent decisions, the rationale of the design process can be captured in a more structured way. Identifying the issues excludes all the irrelevant concerns and makes the progress more focused on the goal to be achieved because, even if the requirement is known, there are many other artifacts that only become apparent as the design progresses. Capturing the relationship between the issues and decisions will provide a good design justification. In this case the designer will be able to see what the 'response' to a particular issue is and which decision has triggered a particular issue. This will minimise the need to 'fake' it (Parnas and Clements, 1986).

In modelling the relationship between design issues and decisions, the AND/OR graph presented in (Jackson, 1990; Shinghal, 1992; Shirai and Tsuji, 1984) is used. The AND notion is used to represent the 'logical' AND while the OR notion is used to represent the 'logical' OR. During a design process sometimes an issue triggers one or more decisions to be made. When there is more than one decision, then these decisions have to be carried out together. Such a condition is represented by the AND notion provided by the AND/OR graph. Similarly, during a design process, a number of decision alternatives may be made. For instance, after making a decision a designer could change his mind and make another one. These decision alternatives are mutually exclusive. That is the designer can only use one of them in any given time. Such a condition is represented by the OR notion in the AND/OR graph.

The AND/OR graph can grow to any depth. For instance, any child node can in turn become an AND or OR node. This condition precisely resembles the decision making process of a designer. In a typical design process, such a graph will become a network representation. Figure 8 illustrates the typical network representation of a design justification made in a design process.

As shown in Figure 8, the relationship between decisions and issues represented as a decision network is quite complex for a human to remember. When the size of a design is large and the design evolves throughout the design process, it is almost impossible for a human -being to keep track of the development details. Losing these design justifications unfortunately will affect the progress of the design process itself. To be able to capture design decisions systematically, a design decision is defined as a set of SDL constructs that are interrelated to each other. These interrelated constructs form a structure which is actually a sub-graph of an intermediate graph representation of a design specification. The relationship between intermediate graph representation and design justification is illustrated in Figure 9.

By maintaining the design justification in this manner, all decisions are captured systematically and the agent is given some degree of flexibility to browse through them. Furthermore, this captured mechanism will allow the agent to record his decisions coherently within a design process. This is a different system from the one that is suggested in Dhar and Jarke (1988) and Potts and Bruns (1988) where the systems design rationales are captured separately.
A large and complex higraph, the overlapping contour makes the higraph itself hard to understand. To overcome this problem, a simple colouring scheme can be adopted.

There is no rigorous and formal notation to represent attributes/properties of an entity. In other words, blobs are used to represent attributes of an object and the object itself. In a large higraph, this could cause some confusion and hence misinterpretation could happen. Simple blob colouring and shaping could be adopted to distinguish the attributes.

Figure 10 illustrates a formal representation model for maintaining the evolution history of SDL-based designs and their justifications. Readers are suggested to refer to (Harel, 1988) for the full discussion of Higraph syntax and terminologies. It should be noted, however, that the formal representation illustrated here only provides a basic framework of how to maintain the SDL-based design specification history. The model does not consider other management information such as personnel name, address, etc. but they can be easily included in the model.

Basically the formal representation model presented consists of three major components: Design State, Design Justification and SDL Object blobs. The design state blob provides a mechanism to capture the evolution history of an object. It is composed of two orthogonal components, they are: StateNo and Revision. The StateNo represents the 'sequence' of the design state's occurrence, for instance: 1, 2, 3, etc. Each of these design states, however, is directly related to a revision that underlies it. This revision is represented by four orthogonal components: RevNo, Description, Delta and Status. The RevNo indicates the revision number. The Description indicates the user's own textual description of the revision, and hence the design state itself. Revisions are represented by deltas, in that they are not stored intact. Therefore one of the orthogonal components of a revision is a delta. The Delta is composed of two sets of SDL objects called &delta; and δ. They represent two sets of objects that need to be added to or removed from a revision. Each of these delta sets is composed of SDL objects. The Status blob (which can be either Current, Alternative, Rejected and Experimental) indicates the status of the revision and hence the design state. The occurrence of a design state is represented by information such as when (date and time) and who (modifier) introduced a design state.

As explained earlier, four type of SDL objects are defined: atomic, link, structure and interface. The SDL Object blob represents these notions. The atomic object is composed of three blobs that represent the properties of the object itself. In other words, atomic objects are modelled as objects that are composed of three attributes: ObjectID, Content and ConsType. ObjectID holds the unique identification of the object itself, Content holds the actual textual content of a construct being represented, and finally, ConsType shows the actual construct being represented.

Link Objects, using the same explanation as the preceding paragraph consist of four attributes: ObjectID, Content, Source, and Destination. ObjectID represents the unique identification
of a link object. Content, as in atomic object, holds the actual textual content of a construct being represented. Since in the model a link object is a directed arc, then there is a need to capture the information as to which is the source and which is the destination node of a link object. Special arcs label $Is_{UNQ.ID.\text{Of}}$ is used to explicitly illustrate the Source and Destination attributes that hold unique identification codes of either a structure or atomic object. This approach is taken to overcome the unnecessary contour overlapping which could make the representation hard to follow.

Structure and interface objects have a similarity in that they have underlying structures which also could have their own design history. Besides this 'speciality', a structure object is similar to an atomic object and an interface object is similar to a link object. Therefore, a structure or interface object is actually an atomic or link object but with additional information about its underlying objects. This notion is represented by UnderlayObject. UnderlayObject is a collection (set) of structures and interface objects. For instance, assume that block B1 uses process P1. In the actual development history, there is a possibility that block B1 at design state 2 uses a revision of process P1 which is 'checked-in' at design state 4. At subsequent design states, block B1 may use a different revision of process P1. The UnderlayObject notion illustrated in the formal model will be able to capture such interrelationships.
Design justification is composed of two orthogonal components, they are Issue and Decision. Each of these design justification elements has its own unique ID and textual description. Furthermore, a design issue can be in a status of either Solved, Unsolved or Rejected. This is represented by the three blobs inside the Status blob. A design issue is raised at a particular design state by a modifier at specific time and date. A modifier can be a designer, maintainer or programmer. An issue can be triggered by a user requirement (which is identified by a requirement ID) or another decision. This is represented by the Is_TriggeredBy relationship.

A design decision is composed of a set of SDL objects that are interconnected to each other forming a logical structure. This is represented by the relation Is_Composed_Of. A decision always exists in response to an issue. A decision also can have alternatives which are identified by decision ID, and this is represented by Is_Alternative_Of.

With this formal representation we are able to capture the evolution history of SDL-based design specifications systematically. Furthermore, the representation model provides a way to capture design justifications in a managed process. Hence the evolution of design artifacts presented in previous sections can be maintained using this formal representation model.

5 SoftEAM, THE DESIGN ARTIFACT MAINTENANCE SYSTEM

SoftEAM, which stands for Software Evolution Assistance and Management, is the name of the prototype tool developed to implement the formal representation model and concepts presented in the previous sections.

This prototype model has been developed under the Microsoft Windows™ environment. This environment is used because it provides a user friendly interface which is needed for the browsing mechanism. At the repository level, a traditional relational database system is used to store the design information. A relational database system is used because it is a common database system, which is transportable to any other similar operating environment. The prototype is designed to provide a portable implementation framework.

The SoftEAM system architecture consists of multiple layers, they are:

— Repository layer. This is the lowest layer in the system. Its purpose is to provide mechanisms to create, delete, read, write and modify units of information. Information such as software objects, design issues, design decisions, design states and so on are stored in this layer.

— Representation layer. This second layer provides logical information. The information stored in the repository level is raw information which is not easily understood by human beings. This layer, therefore, provides the mechanism to logically link those units of information into semantically understandable information. This layer uses services provided by the repository layer.

— Application layer. This highest layer provides applications to use the services provided by the two lower layers. This layer is composed of three sub-systems, they are: browser, specification constructor, and delta generator. Browser provides a mechanism to navigate through all the information stored in the system by utilising the representation layer. The specification constructor provides a mechanism to generate a complete specification of a software design. In cases where there are many design states for a structure/ interface object, the design state tagged with current status will be used to generate the underlying structure. The delta generator will accept two sets of SDL objects as its input and generates deltas. In generating the deltas, this generator will 'ask' the specification generator to produce the specification stored at the last design state. Upon receiving this information and the input from the user, deltas will be generated. The deltas with other information such as design state, status of the design state, etc. are then stored in the repository.

The architecture of SoftEAM is graphically illustrated in Figure 11:

6 CONCLUSION

We have presented a formal representation model to capture the evolution of SDL-based design specifications and design justifications involved in a design process. We have also presented the architecture of our prototype tool called SoftEAM, which was developed to implement our formal representation model and concepts described earlier. We believe that our notion of a maintenance system at object level and systematic capture of design justifications are more appropriate to current software engineering environments. However, we acknowledge that the work we presented here is just the first step towards bringing a new maintenance notion to the software engineering environment. We have restricted our model to maintaining the evolution of SDL-based design specifications. However, the model can be generalised such
that it can be used in other environments. We also intend to enhance the model in several areas such as space and speed optimisation in the information repository and support for the evolution of abstract data types.

REFERENCES


BIOGRAPHICAL NOTES

Glenn Tesla Jayaputera is a software design engineer at Datum Management Systems. In 1993, he completed a Masters of Applied Science in Computer Science at the Royal Melbourne Institute of Technology. His current research interests include formal specification techniques, software process modelling and compiler techniques.

Kong Eng Cheng is a Lecturer in the Department of Computer Science at the Royal Melbourne Institute of Technology. His current research interests include software engineering support environments for formal methods, formal modelling and analysis of intelligent network services.
Software Reusability in Australia

Richard Verschoor
Graham Low
School of Information Systems,
The University of New South Wales,
Sydney, NSW 2052
Email: G.Low@unsw.edu.au

1 INTRODUCTION
Research into software reusability is considered to have started in 1969 when McIlroy published a paper on software reuse: “Mass Produced Software Components” (Tracz, 1988a). Research conducted in the 1970s and early 1980s highlighted the importance of software reusability. For instance, 40-60% of the software analysed at the Raytheon Company was found to be redundant (Lanergan and Poynton, 1981; Lanergan and Grasso, 1984). Software development over a period of three years resulted in levels of code reuse of 42% and 60%. Other case studies, such as BTG, Digital Equipment Corporation, the RAPID project, and Hartford Insurance highlighted the benefits resulting from a programme of software reuse (Incorvaia et al., 1990).

It is a common assertion that reusability can contribute significantly in attaining higher levels of productivity and quality in software development (Basili and Rombach, 1991; Biggerstaff and Richter, 1987; Bollinger and Pfleeger, 1990; Caldiera and Basili, 1991). While research has highlighted the potential of software reuse its promise has been largely unfulfilled (Biggerstaff and Richter, 1987). Little research, however, has been undertaken to investigate the degree to which organisations are actively employing a policy of software reuse.

Various barriers to adopting a policy of software reuse have been suggested. Recent research has suggested non-technical barriers, such as managerial and measurement problems of reuse (Hall, 1987; Isoda, 1992; Tracz, 1988b). It is appropriate to now validate this change in emphasis by examining the experience of Australian organisations. For the purpose of this paper software reuse will refer more specifically to the reuse of prior knowledge and effort associated with program code. The issues considered will be:
- the extent of software reuse in Australian organisations;
- the reasons for not adopting a reuse strategy;
- the relative importance of reuse in a software engineering environment;
- the perceived benefits of software reuse;
- the methods of software reuse currently being used;
- the benefits that each of the current methods of software reuse provide;
- the problems in pursuing a policy of software reuse;
- reuse identification strategies;
- organising software reuse;
- strategies for costing reuse; and
- measuring software reuse.

2 METHOD OF RESEARCH
This research focuses on the extent to which organisations have adopted strategies for organising, managing, and measuring software reuse. As with any study investigating general ‘state-of-practice’, a survey is a feasible means of providing data with sufficient external validity (McGrath et al., 1983).
2.1 The survey sample
The survey was mailed to a total of 228 organisations. Every effort was made to obtain a representative sample to allow for some generalisation of Australian organisations. The organisations were selected from a variety of sources, including:
— 99 from a general company mailing list, with no apparent selection bias;
— 28 from the company sponsors of the Business Information Technology program at the University of New South Wales;
— 10 distributed through Masters Students at the University of New South Wales; and

There were approximately equal proportions of large, medium, and small sized organisations included in the mailing list. The sample of companies represented the banking and finance, manufacturing, information technology, and service industries.

2.2 Survey sample
The survey sent to managers included a cover letter stressing the importance of the research, a self-addressed postage-paid envelope, and the 12 page questionnaire.

The survey instructions required that the questionnaire be completed by the Information Technology Manager or by the Applications Development Manager. Respondents could remain anonymous if desired. In an attempt to encourage organisations to respond, and to provide some motivation for ensuring a greater quality of response, the results were made available to those who completed the survey.

2.3 Survey validation
The survey was validated by 2 managers of one organisation, as well as by 8 Business Information Technology Honours Students. The organisation chosen to perform the validation was familiar with the concepts of software reuse. Following the survey validation minor modifications were made.

2.4 Survey details
No previous survey on software reuse was found. The survey comprised a total of 14 questions. Six of these questions utilised a seven point Likert-type scale. The remaining questions were of a qualitative nature. The 14 questions are briefly outlined.

Question 1 aimed to establish whether Australian organisations consider software reuse to be a worthwhile issue in the context of the variety of other software development issues. The issues were based on Boehm’s (1987) productivity improvement factors. In addition, the issues of Fourth Generation Languages and Object Oriented Programming were included. It was expected that this should provide an insight into how these current issues are viewed in the context of software reusability. The major thrust of Object Oriented Programming, for example, is improved productivity and quality through reusability.

The purpose of question 2 was to establish the extent to which software is being reused in organisations. Based on the answer to this question, respondents were categorised by whether software was reused, whether there existed a policy of software reuse, and whether this policy was ‘organisation-wide’ or ‘project-specific’. The definition of a ‘policy of software reuse’ provided in the survey was “the active endorsement by managers and leaders to reuse software components”. The answer to this question determined which of the subsequent questions would be completed.

Question 3 was completed only by those respondents who indicated in question 2 that they did not have a policy of software reuse. The purpose of this question was to determine why managers have not pursued a policy of reuse given the predicted gains suggested in the literature, and to examine whether the underlying basis for these decisions was of a technical, managerial or behavioural nature. The reuse problems given by Frakes and Gandel (1990) (which are examined in question 12) are aggregated in this question on a technical and managerial basis. The inclusion of an ‘other’ category made it possible to find out any other problems organisations may have encountered.

The purpose of Question 4 was to establish the methods of software reuse currently being utilised by Australian organisations, and the relative perceived benefits of these methods. It is based on the Biggerstaff and Richter (1987) characterisation of reusable technologies.

Question 5 aimed to discover the extent to which reuse identification strategies are being used by organisations. The wide variety of strategies identified in the literature were included (Caldiera and Basili, 1991; Hall, 1987; Kang, 1987; Karimi, 1990; Neighbors, 1984; Prieto-Diaz, 1990).

The purpose of question 6 was to verify the general assumption of implementation productivity and quality benefits of software reuse made in the literature. While question 4 examined the origin of the perceived benefits, question 6 provides an aggregate measure of the nature of these benefits.

A qualitative response was required for questions 7 through 11. The provision for a ‘free-form’ text response is appropriate since the nature of these responses was not well defined in the literature, and the general responses to these questions could not be predetermined. Question 7 examined the extent to which organisation methodologies have been adapted to incorporate reusability, and the ways in which this was done. The importance of this question was highlighted by Hall (1992) and Basili and Rombach (1991). Questions 8 through 11 examined the nature of any infrastructure put in place to facilitate software reuse. This included costing, incentive, and measurement mechanisms described in the literature. The questions were worded in a manner as to prompt a response that would highlight the organisation’s activities that were consistent with the literature.

Question 12 lists the reuse problem categories presented by Frakes and Gandel (1990). While the literature presents a variety of opinions as to why reuse “has never acquired real
SOFTWARE REUSABILITY IN AUSTRALIA

momentum” (Caldiera and Basili, 1991), little is known about the extent of the problems listed. Question 12 requires the respondent to consider the major problems confronting an organisation contemplating a policy of software reuse.

3 Results and discussion

3.1 Questionnaire results

A total of 61 surveys were received. Of the 61 completed and uncompleted surveys:

— 46 were valid responses;
— 12 were received as ‘return to sender’ presumably due to inaccuracies in the mailing list; and
— 3 surveys received were incomplete.

The 46 valid surveys received reflects a valid response rate of 20.2%. This is considered sufficient for this research because of no apparent selection bias in the sample of valid surveys received. An analysis of the responding organisations reveals that there are approximately equal number of firms in each industry group, and a relatively similar proportion of small, medium, and large sized firms.

The general quality of the surveys received increases the adequacy of the response. This quality is reflected in a review of the job titles of the respondents, which suggest that over 90% of the surveys were completed by senior information technology managers. Forty-four of the forty-six responding organisations reused software.

3.2 The extent of software reuse

The extent to which software is reused by organisations is shown in Table 3-1. It was found that about 50% of the respondents have a policy of software reuse.

— 12 organisations (about 26%) have a policy of reuse which applies to software development across the organisation;
— 13 organisations (about 28%) have a policy of reuse which applies in one or more projects;
— 19 organisations (about 41%) do not have a software reuse policy, but reuse, nevertheless, does take place within projects; and
— only 2 organisations (about 5%) claim that no reuse takes place within projects.

Table 3-1: The Extent of Software Reuse.

<table>
<thead>
<tr>
<th>Policy Reuse</th>
<th>Extent</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>Organisation Wide</td>
<td>12</td>
</tr>
<tr>
<td>✓</td>
<td>1 or More Projects</td>
<td>13</td>
</tr>
<tr>
<td>✗</td>
<td>1 or More Projects</td>
<td>19</td>
</tr>
<tr>
<td>✗</td>
<td>None</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 3-2: Reasons For Not Adopting a Policy of Software Reuse.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Median</th>
<th>Mode</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Never Considered</td>
<td>3</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>Difficulty in managing a software engineering environment, promoting and rewarding reuse</td>
<td>5</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>The unsuitability, or lack of infrastructure within the organisation</td>
<td>6</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>Difficulty involved in designing software for reuse</td>
<td>6</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>Lack of technology within the organisation</td>
<td>6</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>Lack of technology in the industry</td>
<td>6</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>Change to a ‘reuse mindset’ is too difficult</td>
<td>7</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>Believe the cost benefits from reuse are not sufficient</td>
<td>7</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>Programmer/Analyst resistance</td>
<td>7</td>
<td>7</td>
<td>3</td>
</tr>
</tbody>
</table>

Contrary to the criticisms of the ‘state-of-practice’ (that organisations are not embracing the reuse paradigm or that reuse is not delivering the expected benefits), it was somewhat surprising to discover that over half of these respondents have a policy of reuse — where managers and leaders actively endorse and support the reuse of software components. Swanson and Curry (1989) and Prieto-Diaz (1991a) note that organisational commitment is essential to effectively reuse software. Half of the organisations surveyed claim to have met this requirement.

The reasons why the remaining 46% of the organisations surveyed have not formally embraced the reuse paradigm was investigated [Table 3-2]. Of the wide variety of reasons suggested in the literature, the main reason given was that software reuse has simply not been considered by these organisations. The respondents did not consider the absence of benefits or excessive costs to be a major reason. Other than not considering reuse, the other major inhibitors to committing to the reuse of software were related to the difficulty in managing a software engineering environment. This finding is consistent with the trend in software reuse research in placing less emphasis on the technical problems of software reuse and greater importance on the managerial aspects. These results concur with the views of Traczc (1988b) and Isoda (1992).

An ‘other’ factor was included in question 3 to capture any reasons for not adopting a policy of reuse that was not identified in the literature. Respondents could suggest another reason and rate the extent to which this was a problem. Two
organisations gave a reason not included in the question. One organisation claimed that the proliferation of packages and further development driven by the system’s architecture was a reason for not adopting a policy of reuse. This is an interesting reason which, with the current increase in the use of packages, has implications for the issue of software reuse. Taking a broader view, however, packages themselves are a powerful means of reuse (Biggerstaff and Richter, 1987). The other reason given was that there exists a “significant investment in an existing portfolio of applications which were never designed for software reusability” (a large international firm in the IT industry). This reason for not adopting a policy of reuse has not been suggested in the literature. It may be a possible explanation for the ‘state-of-practice’. It is apparent that many of the organisations reusing software have “just commenced the philosophy” (a large sized firm in the manufacturing and services industry). This finding is unexpected, considering that reuse concepts were initiated in 1969. This may suggest that the state-of-practice of software reuse is still in its infancy.

3.3 The issue of software reuse

Table 3-3 ranks the extent to which software reuse is considered an important software development issue. Various factors identified in the literature as capable of making an improvement in software development productivity were listed. The median, mode and range are given for each factor. The results have been grouped according to whether or not the organisation has a policy of reuse (with sample sizes of 25 and 19 respectively). Those organisations with a reuse policy are grouped further by whether this policy is organisation or project-wide (with sample sizes 12 and 13 respectively). The ranking of factors (based on the median and mode) is given for each group. A Kruskal-Wallis test was also performed to determine whether the three groups are significantly different for each of the factors. The Chi-Square and significance for each factor are given in the table.

Of the 10 various issues listed, software reusability was rated between 4 and 6 depending on the group. All groups agree that acquiring and getting the best from personnel, project management, and programmer and analyst training are the three main issues. The results indicate that the three groups are not significantly different for any factor at the 0.05 level. Although there is a slight difference in the ranking of software reusability between the groups, the difference in the mean has the lowest Chi-Square value (of 0.0064) of all the factors. This may indicate that the importance of the issue of software reusability is well established.

<table>
<thead>
<tr>
<th>Issue</th>
<th>Policy of Reuse</th>
<th>No Reuse Policy</th>
<th>Kruskal-Wallis Chi-Square (significance)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Organisational Reuse Policy</td>
<td>'Project-wide' Reuse Policy</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Median Rank Mode Range</td>
<td>Median Rank Mode Range</td>
<td>Median Rank Mode Range</td>
</tr>
<tr>
<td>Project management</td>
<td>1 2 (4)</td>
<td>1 2 (4)</td>
<td>=1 2 (3)</td>
</tr>
<tr>
<td>Staffing: acquiring and getting best from personnel</td>
<td>2 2 (3)</td>
<td>1 1 (2)</td>
<td>1.01 (0.61)</td>
</tr>
<tr>
<td>Programmer/Analyst training</td>
<td>3 2 (2)</td>
<td>1 2 (4)</td>
<td>=2 2 (4)</td>
</tr>
<tr>
<td>Software methodologies</td>
<td>4 2.5 (5)</td>
<td>2 2 (4)</td>
<td>=4 2 (4)</td>
</tr>
<tr>
<td>Software reusability</td>
<td>5 2.5 (4)</td>
<td>3 3 (5)</td>
<td>=6 3 (5)</td>
</tr>
<tr>
<td>Prototyping</td>
<td>6 3 (4)</td>
<td>4 2 (5)</td>
<td>=4 2 (5)</td>
</tr>
<tr>
<td>Fourth Generation Languages</td>
<td>=7 3.5 (2)</td>
<td>=6 3 (5)</td>
<td>=6 3 (5)</td>
</tr>
<tr>
<td>Automated Documentation and Quality Assurance</td>
<td>=7 3.5 (2)</td>
<td>=6 3 (5)</td>
<td>=6 3 (5)</td>
</tr>
<tr>
<td>Object Oriented Programming</td>
<td>9 4 (5)</td>
<td>=6 3 (5)</td>
<td>10 4 (5)</td>
</tr>
<tr>
<td>CASE tools, software tools, productivity environment</td>
<td>10 4 (4)</td>
<td>=6 3 (4)</td>
<td>=6 3 (6)</td>
</tr>
</tbody>
</table>

Table 3-3: Software Reuse as an Information Systems Issue.
3.4 Perception of the benefits of software reuse

No organisation was able to adequately quantify the costs and benefits in reusing software. Organisations generally, however, perceived substantial benefits from software reuse [Table 3-4].

The only factor where there was a significant difference between the three groups was "improved productivity in the coding stage" (significance of 0.04). There was some diversity in the relative importance of the variety of benefits between the respondents. Software appears to be reused for more than simply improving productivity. This finding is important since the consideration of productivity as the primary motivation for reuse is a general assumption made in the literature. It also suggests that it is not possible to evaluate the general success of software reuse by solely considering the resulting productivity improvements.

The respondent of a large sized firm in the banking and finance industry, with a policy of reuse, remarked that "reusability is a two edged sword — the more times an object is reused, the more difficult it is to change". The reuse model accepted at this organisation forms a triangle with wide reusability, low complexity, and high efficiency. "If you plot a point between these goals you realise reuse entails a tradeoff. Reusing objects often results in systems with greater complexity and lower efficiency". This criticism of reuse needs further investigation.

3.5 Methods of software reuse

Question 4 required respondents to rate the extent to which they used various methods of reusing software. The results, Table 3-5, suggest that the more traditional or fundamental methods of code reuse (such as modules, copybooks, and higher level languages) are more widely used than methods which facilitate reuse (such as program generators and object oriented languages). There was a significant difference (significance of 0.03) between the three groups for "higher level programming languages". Using the Biggerstaff and Richter (1987) characterisation of reusability technologies framework, organisations generally appear to be using the technologies which offer a compromise between 'power' and 'generality'.

3.6 Benefits of software reuse methods

The perceived benefit of software reuse for each reuse method is given in Table 3-6.

Table 3-6 shows that most organisations perceive that there is benefit in utilising the methods of reuse listed. The first seven methods in each group all have a median of less than or equal to 3.5. Generally, methods are used according the extent to which benefits are perceived. The difference between the extent of utilisation and benefit scales was calculated and examined for each factor. It was shown that generally the utilisation and benefit scales are orthogonal.

The results provide support for the Biggerstaff and Richter (1987) reusability technologies framework. The benefits from the methods may be considered to be the 'power' and the extent of utilisation the 'generality'. Program generators, for example, had one of the highest benefit ratings, but one of the lowest utilisation ratings. The findings presented concur with the "generality of applicability versus payoff" dilemma described by Biggerstaff and Richter (1987).

Table 3-4: Perceived benefits of Software Reuse.
The
AUSTRALIAN COMPUTER JOURNAL
VOLUME 26
February 1994 to November 1994

CONTENTS

Contributions

1. February, 1994
Blair, A. Inter-Connection of Software Components in Intelligent Decision Support Systems 11-19

2. May, 1994
Wijegunaratne, L, Socic, M. and Chow, B. An Architecture for Client/Server Application Software 30-41
Krishnan, P. Specification of Schedules Using the Modal-μ Calculus 42-49
Goscinski, A. and Haddock, A. A Naming and Trading Facility for a Distributed System 50-65

3. August, 1994
O'Connor, K. Confidentiality, Privacy and Security Concerns in the Modern Healthcare Environment 70-77
deSilva, C.J.S., Choong, P.L. and Attikiouzel, Y. Artificial Neural Networks and Breast Cancer Prognosis 78-81
Thornton, B.S., Bloomell, C. and Thornton, F. Comparison of Centralised and Decentralised CT Radiology Services using Separated Scan Heads, IT and High Speed Communication Links — A New Telemedicine Study 82-89
Morrison, M. and Attikiouzel, Y. An Introduction to Segmentation of Magnetic Resonance Medical Images 90-98
Louw, G. Patient Focused Care — Self Contained Units Reduce the Need for Information Processing 99-101
Rasiah, A.I. and Attikiouzel, Y. A Syntactic Approach to the Recognition of Common Cardiac Arrhythmias within a Single Ambulatory ECG Trace 102-112

Jayaputera, G.T. and Cheng, K.E. SOFTERAM: A Design History and Justification Maintenance Tool 124-133

Verschoor, R. and Low, G. Software Reusability in Australia 134-142

Slade, G. and Parker, N. SEE-Ada: Software Evaluation Environment for Ada 143-150


Hoffman, D. and Strooper, P. Graph-based Class Testing 158-163

Miscellaneous

Brisbane Summer Technical Conference 10

Call for Papers — AUUG-’94 10

Letters to the Editor 20, 66

Books Received 21, 67

Position Vacant — Associate Editor for Book Reviews 21

Book Reviews 22-28, 68, Cover III, 164-Cover III

Editorial 29, 69, 113

Appointment of a New Editor 65

Appointment of a New Associate Editor for Book Reviews 65

Call for Papers — Parallel Processing: Algorithms, Architectures and Applications 66
A Naming and Trading Facility for a Distributed System 50-65
A Syntactic Approach to the Recognition of Common Cardiac Arrhythmias within a Single Ambulatory ECG Trace 102-112
An Architecture for Client/Server Application Software 30-41
An Introduction to Segmentation of Magnetic Resonance Medical Images 90-98
Artificial Intelligence and Mathematical Theory Computation 22
Artificial Neural Networks and Breast Cancer Prognosis 78-81
Automated Reasoning: Essays in Honour of Woody Bledsoe 22-23
Comparison of Centralised and Decentralised CT Radiology Services using Separated Scan Heads, IT and High Speed Communication Links — A New Telemedicine Study 82-89
Compiling Parallel Loops for High Performance Computers 25-26
Compression and Fast Indexing for Multi-Gigabyte Text Databases 1-9
Computer Society Reference Book 28
Computer Vision and Image Processing 68
Computerised Accounts (Third Edition) Cover III (Nov)
Confidentiality, Privacy and Security Concerns in the Modern Healthcare Environment 70-77
Data Communications Testing and Troubleshooting 23-24
Data Fusion in Robotics and Machine Intelligence 24
Distributed Artificial Intelligence: Theory and Praxis 25
Distributed Computing: A Practical Synthesis Cover III (Nov)
Enterprise Information Technologies Cover III (Aug)
Essentials of Logic Programming 27-28
Exploiting Chaos — Cashing in on the Realities of Software Development 24-25
Foundations of Knowledge Acquisition: Machine Learning 26-27
Generating Abstraction Hierarchies: An Automated Approach to Reducing Search in Planning 26
Graph-based Class Testing 158-163
Information Engineering: Strategic Systems Development 68
Inter-Connection of Software Components in Intelligent Decision Support Systems 11-19
Mathematical Modeling 24
Parallel Algorithms and Architectures for DSP Applications 23
Patient Focused Care — Self Contained Units Reduce the Need for Information Processing 99-101
PC Magazine DOS 6 Techniques & Utilities Cover III (Aug)
Physically-Based Modelling for Computer Graphics Cover III (Aug)
Practical Neural Network Recipes in C++ 27
Quality Management for Software 164
Quality Management within PRINCE 68
Running MS-DOS 164-Cover III (Nov)
Scalable Shared Memory Multiprocessors 22
Scientific Computing: An Introduction with Parallel Computing 164
SEE-Ada: Software Evaluation Environment for Ada 143-150
Simple Program Design 25
SotFEAM: A Design History and Justification Maintenance Tool 124-133
Software Production Management Cover III (Nov)
Software Reusability in Australia, 134-142
Specification of Schedules Using the Modal-μ Calculus 42-49
Surviving Paradigm Shifts in Software Development Technology: A Management Case Study of Industrial Experience 114-123
TEXHelp: The On-line TEX Handbook 27
The Computer Artist’s Handbook 24
What is DIP? — A Guide to Document Image Processing 164
What is EDI? — A Guide to Electronic Data Interchange (2nd Edition) 164
<table>
<thead>
<tr>
<th>Contributor</th>
<th>Page Numbers</th>
<th>Additional Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anderson, M.</td>
<td>28</td>
<td>Louw, G. 99-101</td>
</tr>
<tr>
<td>Attikiouzel, Y.</td>
<td>78-81, 90-98, 102-112</td>
<td>Low, G. 134-142</td>
</tr>
<tr>
<td>Bailes, P.A.</td>
<td>151-157</td>
<td>Martin, E. 27-28</td>
</tr>
<tr>
<td>Bartlett, J.</td>
<td>68</td>
<td>McCormack, J. 24</td>
</tr>
<tr>
<td>Blair, A.</td>
<td>11-19</td>
<td>McGrath, T. 164</td>
</tr>
<tr>
<td>Bloesch, A.</td>
<td>24</td>
<td>Moffat, A. 1-9</td>
</tr>
<tr>
<td>Bloomell, C.</td>
<td>82-89</td>
<td>Morrison, M. 90-98</td>
</tr>
<tr>
<td>Breen, J.</td>
<td>23-24</td>
<td>O'Connor, K. 70-77</td>
</tr>
<tr>
<td>Chapman, M.</td>
<td>151-157</td>
<td>Oudshoorn, M. Cover III (Nov)</td>
</tr>
<tr>
<td>Cheng, K.E.</td>
<td>124-133</td>
<td>Parker, N. 143-150</td>
</tr>
<tr>
<td>Chia, M.</td>
<td>151-157</td>
<td>Peake, I. 151-157</td>
</tr>
<tr>
<td>Choong, P.L.</td>
<td>78-81</td>
<td>Rasiah, A.I. 102-112</td>
</tr>
<tr>
<td>Chow, B.</td>
<td>30-41</td>
<td>Ray, S. 68</td>
</tr>
<tr>
<td>Colomb, R.M.</td>
<td>25, 26</td>
<td>Riddiford, M. Cover III (Nov)</td>
</tr>
<tr>
<td>Connaughton, C.</td>
<td>114-123</td>
<td>Rout, T. 164</td>
</tr>
<tr>
<td>Crawford, J.</td>
<td>22</td>
<td>Schmidt, H.W. 164</td>
</tr>
<tr>
<td>Dampney, C.N.G.</td>
<td>114-123</td>
<td>Slade, G. 143-150</td>
</tr>
<tr>
<td>deSilva, C.J.S.</td>
<td>78-81</td>
<td>Socic, M. 30-41</td>
</tr>
<tr>
<td>Forbes, L.K.</td>
<td>27</td>
<td>Sonenberg, L. 22, 22-23</td>
</tr>
<tr>
<td>Fraser, D.</td>
<td>23</td>
<td>Stevenson, A. Cover III (Aug), 164-Cover III (Nov)</td>
</tr>
<tr>
<td>Freeman, G.</td>
<td>Cover III (Aug)</td>
<td>Strooper, P. 158-163</td>
</tr>
<tr>
<td>Goold, A.</td>
<td>25</td>
<td>Thornton, B.S. 82-89</td>
</tr>
<tr>
<td>Goscinski, A.</td>
<td>50-65</td>
<td>Thornton, F. 82-89</td>
</tr>
<tr>
<td>Haddock, A.</td>
<td>50-65</td>
<td>Verschoor, R. 134-142</td>
</tr>
<tr>
<td>Hanson, L.</td>
<td>24-25</td>
<td>Wheeldon, A. Cover III (Aug)</td>
</tr>
<tr>
<td>Haywood, L.</td>
<td>Cover III (Nov)</td>
<td>White, M. 68</td>
</tr>
<tr>
<td>Hoffman, D.</td>
<td>158-163</td>
<td>Wijegunaretne, I. 30-41</td>
</tr>
<tr>
<td>Jacob, A.P.</td>
<td>27</td>
<td>Wu, X. 26-27</td>
</tr>
<tr>
<td>Jayaputra, G.T.</td>
<td>124-133</td>
<td>Zhang, K. 25-26</td>
</tr>
<tr>
<td>Kewley, D.</td>
<td>24</td>
<td>Zobel, J. 1-9</td>
</tr>
<tr>
<td>Krishnan, P.</td>
<td>42-49</td>
<td></td>
</tr>
</tbody>
</table>
### Table 3-5: Methods of Software Reuse.

<table>
<thead>
<tr>
<th>Method</th>
<th>Policy of Reuse</th>
<th>No Reuse Policy</th>
<th>Kruskal-Wallis Chi-Square (significance)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Organisational Reuse Policy</td>
<td>'Project-wide' Reuse Policy</td>
<td>Rank</td>
</tr>
<tr>
<td></td>
<td>Rank</td>
<td>Median</td>
<td>Mode</td>
</tr>
<tr>
<td>Copybooks</td>
<td>1</td>
<td>2 (3)</td>
<td>2</td>
</tr>
<tr>
<td>Libraries of Modules</td>
<td>2</td>
<td>2.5 (1)</td>
<td>3</td>
</tr>
<tr>
<td>Program Skeletons</td>
<td>3</td>
<td>3 (6)</td>
<td>5</td>
</tr>
<tr>
<td>Application Frameworks and Architectures</td>
<td>4</td>
<td>3 (6)</td>
<td>6</td>
</tr>
<tr>
<td>Report/Screen Generators</td>
<td>5</td>
<td>4 (5)</td>
<td>=2</td>
</tr>
<tr>
<td>Higher Level Programming Languages</td>
<td>6</td>
<td>4.5 (5)</td>
<td>1</td>
</tr>
<tr>
<td>Program Generators</td>
<td>=7</td>
<td>7 (6)</td>
<td>=7</td>
</tr>
<tr>
<td>Program Logic Structures</td>
<td>=7</td>
<td>7 (6)</td>
<td>=7</td>
</tr>
<tr>
<td>Object Oriented Programming Languages</td>
<td>=7</td>
<td>7 (4)</td>
<td>=7</td>
</tr>
</tbody>
</table>

### Table 3-6: Benefits From Methods of Software Reuse.

<table>
<thead>
<tr>
<th>Method</th>
<th>Policy of Reuse</th>
<th>No Reuse Policy</th>
<th>Kruskal-Wallis Chi-Square (significance)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Organisational Reuse Policy</td>
<td>'Project-wide' Reuse Policy</td>
<td>Rank</td>
</tr>
<tr>
<td></td>
<td>Rank</td>
<td>Median</td>
<td>Mode</td>
</tr>
<tr>
<td>Copybooks (39 cases)</td>
<td>=1</td>
<td>2 (3)</td>
<td>3</td>
</tr>
<tr>
<td>Program Generators (16 cases)</td>
<td>=1</td>
<td>2 (5)</td>
<td>3</td>
</tr>
<tr>
<td>Libraries of Modules (39 cases)</td>
<td>=3</td>
<td>2 (3)</td>
<td>=1</td>
</tr>
<tr>
<td>Application Frameworks and Architecture (35 cases)</td>
<td>=3</td>
<td>2 (04)</td>
<td>=6</td>
</tr>
<tr>
<td>Program Skeletons (37 cases)</td>
<td>=5</td>
<td>2.5 (5)</td>
<td>=6</td>
</tr>
<tr>
<td>Higher Level Programming Languages (35 cases)</td>
<td>=5</td>
<td>2.5 (5)</td>
<td>=1</td>
</tr>
<tr>
<td>Report/Screen Generators (33 cases)</td>
<td>7</td>
<td>3 (1)</td>
<td>3</td>
</tr>
<tr>
<td>Object Oriented Programming Languages (7 cases)</td>
<td>8</td>
<td>4.5 (3)</td>
<td>=6</td>
</tr>
<tr>
<td>Program Logic Structure (16 cases)</td>
<td>9</td>
<td>5 (1)</td>
<td>9</td>
</tr>
</tbody>
</table>
### Table 3-7: Software Reuse Policy Problems.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Policy of Reuse</th>
<th>No Reuse Policy</th>
<th>Kruskal-Wallis Chi-Square (significance)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Median Range</td>
<td>Median Range</td>
<td>Median Range</td>
</tr>
<tr>
<td></td>
<td>Rank</td>
<td>Rank</td>
<td>Rank</td>
</tr>
<tr>
<td>Design problems</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Representation and</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>identification problems</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Managerial problems</td>
<td>4</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Economic problems</td>
<td>4</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Measurement and</td>
<td>5</td>
<td>4.5</td>
<td>4</td>
</tr>
<tr>
<td>evaluation problems</td>
<td>6</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>Storage and retrieval</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>problems</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>Legal problems</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>

### Table 3-8: Utilisation of Reuse Identification Strategies Grouped by Extent of Reuse.

<table>
<thead>
<tr>
<th>Identification Strategy</th>
<th>Policy of Reuse</th>
<th>No Reuse Policy</th>
<th>Kruskal-Wallis Chi-Square (significance)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Median Range</td>
<td>Median Range</td>
<td>Median Range</td>
</tr>
<tr>
<td></td>
<td>Rank</td>
<td>Rank</td>
<td>Rank</td>
</tr>
<tr>
<td>No particular strategy</td>
<td>1</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Individual project</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>groups</td>
<td>3</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>Application framework</td>
<td>4</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>Top down approach</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>Asset management group</td>
<td>6</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>Domain analysis</td>
<td>7</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Use of metrics and</td>
<td>8</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>models</td>
<td>9</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>
3.7 Problems in pursuing a policy of software reuse
An understanding of the problems in reusing software is necessary in determining the future direction of research efforts. For example, the perception that there are significant problems associated with the storage and retrieval of reusable components has resulted in considerable research in this area in the past.

Numerous problems have been cited by practitioners and researchers. A categorisation of reuse problems was given by Frakes and Gandel (1990). Frakes and Gandel were unable, however, to ascertain the relative importance of the problems. The findings of this study assist in determining their relative importance.

The result of the survey [Table 3-7] found that the major problems experienced by organisations are related to the design, representation and identification of components, and managerial problems.

The most important problem was in designing software so that it is reusable. This result is significant considering that “a number of studies suggest that design for reusability is the only way software designers can approach an order-of-magnitude increase in productivity or quality” (Karimi, 1990).

While the organisational reuse group appeared to rate certain problems as being more important than either of the other groups, no significant differences were found.

Two issues not included in the survey were also raised by the organisations. The first related to reusing software within a non-homogeneous application environment — “each package has a different set of standards and coding structures”. The other problem was the ability to update reusable components for new requirements, while maintaining support for previous users of these components.

3.8 Reuse identification strategies
The importance of the need for a strategy to identify reusable components has been stressed in the literature (Caldiera and Basili, 1991; Prieto-Diaz, 1991a). However those organisations which did not have a policy of reuse did not utilise the strategies to any great extent. Surprisingly many respondents with an organisation-wide reuse policy claimed to have no particular identification strategy. Fewer organisations with a project-wide reuse policy, however, claimed that they do not have a strategy for identifying reusable components.

3.9 Organising software reuse
The survey investigated the strategies being used to organise the software reuse environment. It was found that over three quarters of the respondents have no formal infrastructure for reuse. Only two respondents claimed to have the complete infrastructure suggested by Prieto-Diaz (1991). Very few organisations had an asset management group or an identification and qualification group.

Karimi (1990) states that many of the technical problems associated with reusability at the design and code level are due to the lack of methodologies and tools for the identification, representation, classification, translation, and composition of reusable software parts. Hall (1992) claims that “the possibility of software reuse can be added to the traditional life-cycle model of software development”. In this regard, it was found that despite the lack of reuse methodologies identified in the literature, certain organisations appear to have successfully adapted their existing methodologies for software reuse. About one third of the respondents claimed to have successfully integrated reusability into their life cycle methodology.

3.10 Strategies for costing reuse
The parallel of software reuse with the capital industry made in the literature suggests that a strategy for accounting for the costs of reuse is most important. Since reuse often requires a long-term, multi-project perspective, with possibly different producers and consumers of reusable components, it is argued that it necessary to account for the up-front investment and costs involved. The survey found that only 2 respondents accounted for the costs of reuse.

3.11 Measuring software reuse
The measurement of reuse is considered to be one of the future developments in the area of software reuse (Hall, 1987). The results found that many organisations also consider this to be an important issue. Of the seven organisations utilising estimations models, only one considers the effect of reuse. Certain organisations aware of this effect either desired to be able to measure reuse to enable a correct interpretation of the metrics and models, or considered the effect to be fundamental to a reuse productivity incentive mechanism. The findings in this study support the recent research emphasis investigating reuse measurement.

4 CONCLUSION
The research objectives of this study were formulated by collating the outstanding research questions identified in the software reuse literature over the past five years. This study has, for the first time, provided some evidence of the current reuse ‘state-of-practice’. Prior claims and allegations of the state of practice have been made without reference to any research. Little research has been undertaken in the past to identify the strategies that have been adopted to promote an efficient and effective program of software reuse, and their associated successes and problems.

This study has determined the extent to which software reuse is considered to be an issue, the extent to which software is currently being reused, the reasons why some organisations have not embraced the reuse paradigm, the degree of success these organisations have had, the methods and techniques for software reuse, the problems that organisations have experienced and the strategies being used to cost, organise and measure software reuse. The results of this study provide a greater general understanding of the practice of software reuse today.
It appears that Australian organisations are currently benefiting from the practice of software reuse. The potential for further future benefit has been recognised. Should the trend to institutionalise reuse claimed by Prieto-Diaz (1991b) continue beyond 'state-of-the-art', Australian organisations may attain even higher levels of software development productivity and quality.

REFERENCES


BIOGRAPHICAL NOTES

Graham Low is a Senior Lecturer at the University of New South Wales. He had in excess of ten years experience in industry prior to joining the School in 1987. His current research interests include reuse, CASE, estimation and telecommunications.

Richard Verschoor completed a BSc (Hons) with a major in Information Systems in 1992. He is currently working for Andersen Consulting.
SEE-Ada: Software Evaluation Environment for Ada

Grant Slade
CSC Australia,
Technology Park,
The Levels, South Australia 5095

Neville Parker
Communications Division,
Defence Science and Technology Organisation,
Salisbury, South Australia 5108

The SEE-Ada tool has been developed to allow the graphical display of the dependency structures of large Ada systems. SEE-Ada uses an Oracle relational database and can import any metric data, capable of conversion to the SEE-Ada standard format. Metric data loaded into SEE-Ada can be displayed with graphical user interface techniques allowing the user ready access to any of the loaded metric data. Integration with the Ada quality metrics analysis tool, AdaMAT/D, is provided. The use of SEE-Ada as an Independent Validation and Verification tool and as an automated Ada development support tool is discussed.

1 INTRODUCTION

SEE-Ada has been developed by the Information Technology Division (ITD), Software Engineering Group of the Defence Science and Technology Organisation (DSTO). Among the consulting functions performed by the Software Engineering Group are the assessment of contractor bids for development of Department of Defence software and the evaluation of software systems delivered to the Department of Defence.

In 1989, the Software Engineering Group began the Software Measurement and Analysis Testbed (SMAT) research project to investigate the problems of quality assessment of large software systems. The Software Evaluation Environment for Ada (SEE-Ada) is the principal product of this research and provides automated support for the analysis of large Ada systems (Vernik et al, 1991). This tool has been used within Software Engineering Group, in conjunction with AdaMAT, an Ada quality metrics tool. SEE-Ada was commercially released at the Australian CASE-Ada Conference in October 1992. SEE-Ada is currently only available for the Sun platform but is being ported to the DEC platform. However, SEE-Ada can analyse Ada code produced by any Ada development system, after the appropriate file format conversion.

Software Engineering Group have evaluated a number of existing commercial tools including Adagen and AdaMAT, and have developed a number of special purpose research tools. These tools have a number of problems, the most severe being their lack of scaleability to large systems (Vernik et al, 1991; Vernik and Turner, 1991). For example, the output of Adagen for a modest sized system of approximately 50 Ada objects is depicted in Figure 1. As can be observed, the graphical display is too cluttered to be useful. A further example is the SPARCworks/Ada AdaVision tool (Sun Microsystems, 1990) which produces a tree structure display of Ada dependencies and this display becomes overwhelmingly large with a large number of units. AdaMAT generates Ada code quality metrics and it too suffers from the problem of scaleability. With large systems AdaMAT produces a voluminous report which is difficult to interpret.

SEE-Ada was designed to provide a graphical view of an Ada system that would: (1) work for systems with large numbers of Ada objects and; (2) allow ready access to Ada code metrics produced by special purpose metric generation tools, including AdaMAT.

The main functions of SEE-Ada will be described briefly and the application of SEE-Ada to the analysis of Ada code quality, both during and after software development, will be discussed.

1 Institute of Computer Systems Engineering and Assurance, University of South Australia, The Levels, South Australia 5095.
2 Mark V Systems Ltd, Encino, California, USA.
3 Dynamics Research Corporation, Andover, Massachusetts, USA.
2 SEE-Ada FUNCTIONS
SEE-Ada provides a series of graphical representations (Views) displaying the static structure of an Ada system. These Views display structural dependency relationships and encapsulation, and allow selective access and display of metric data within these Views using point and click techniques. SEE-Ada Views may be saved and printed if required.

The system architecture of SEE-Ada is depicted in Figure 2. Code is parsed and structural information loaded into the Oracle relational database. Code may be independently analysed by external analysis tools, e.g. AdaMAT/D, the output converted to a standard file format by an appropriate Import Filter Tool and the data loaded into the Database. A series of different architectural views are provided, built from information contained in the Database. Figure 3 depicts the object representation used in the SEE-Ada views.

Layers view
An Ada dependency graph is difficult to comprehend for all but trivial systems, as shown by the Adagen output in Figure 1. For large systems with hundreds or thousands of objects, an effective means of representing dependency relationships is required.

Figure 4 shows the mapping of the Ada dependency graph to the virtual layers of the SEE-Ada Layers View. The arrows in Figure 4(a) indicate dependencies (with clauses), as shown by the following code fragment of package A:

```ada
with B;
with D;
with G;
package A is...
```

The virtual layers (columns) shown in Figure 4(b) are constructed from dependency relationships. Objects in any given layer have dependencies to objects in layers to the right of the current layer. Where cyclic dependencies exist between objects (dependency clusters), these objects are mapped to the same column. The first layer should contain one object, the main procedure. The presence of additional objects in the first layer indicates unused objects.
Figure 2. Architecture of SEE-Ada. The figure depicts the architecture of SEE-Ada. The Parser, Oracle Database, Architecture View Generators and Analysis Tools are shown within the SEE-Ada tool (grey region). Import Filter Tools are utilities run independently of SEE-Ada producing output in SEE-Ada standard file format.

Figure 3. Ada Object Representation. The figure displays the representation of Ada objects within SEE-Ada. Normal Ada objects are shown in solid outline, generic units in dashed outline and null units (code missing) in dotted outline. Subunits, subprograms or tasks are narrower than their corresponding object type.

Figure 5 depicts a screen layout of the Layers View for a small demonstration system Program_Metrics of approximately 16,500 source lines of code (SLOC). In the first column the main procedure specification and body can be seen. In addition, an unused package specification and body, and an unused definition package specification are visible. Several packages with sub-units and several generic packages may also be seen. Sub-units appear in a Layers View only if the sub-units have additional dependencies to their parents. Several null objects (source code not present), representing standard Ada library packages (Text_Io, Calendar and Direct_Io), may also be seen.

Cluster analysis
The ninth column (Figure 5) contains 23 packages, and its length relative to adjacent columns suggests that a dependency cluster may exist among the packages of that layer, as objects with cyclic dependencies map into a single column. A cluster is a group of Ada objects with a least one dependency cycle. The cluster is shown in the Graph View window overlying the Layers View window. By default, the Graph View separates package specification and body, and equi-spaces all components around the circumference of a circle. A facility exists for moving the position of component objects within the Graph View to display dependencies more clearly, and this facility has been used in this example. The cluster contains the Parsetables specification and

Parsetables body, and 21 definition specifications. All definition specifications are dependent on the Parsetables specification, and the Parsetables body is dependent on all the definition specifications. Dependencies also exist between the definition specifications themselves. In all probability, the definition specifications contain type definitions and constant declarations. This cluster presents no apparent cause for concern; however, clusters frequently indicate poor design, and suggest potential compilation and maintenance problems.

Figure 6 is a hypothetical example of a deleterious cluster. This cluster consists of five package specifications and bodies. A number of a cycles exist in this example with the largest being: A body → B spec; B body → C spec; C body → D spec; D body → E spec; and E body → A spec. An example of a small cycle is A body → B spec; B body → A spec. A significant number of additional dependencies are present in this example making a complex dependency relationship between the component packages. While determining a possible compilation order for packages in this particular cluster is not difficult, this complexity might be reduced to a large extent. Much of this type of problem is caused by either: obsolete context clauses (with clauses) remaining from development or testing; or sub-optimal allocation of subprogram, type, variable and constant declarations among packages.

Cluster analysis finds the largest possible cycle. Clusters have been found in real systems containing in excess of 30 packages with more complex interdependency relationships than depicted in this example. Such clusters pose serious maintenance problems where enhancements to these systems are required.
Figure 5. Layers View. The figure displays a SEE-Ada screen showing a Layers View of a small Ada system Program_Metrics of approximately 16,500 SLOC. The Test_Tools body has been selected and the objects highlighted are those objects that Test_Tools ‘With’s. When colour is used the highlighting would be in a lighter shade of the selected object colour. The tracing feature is also shown. A forward trace has been constructed to show a sequence of dependencies starting at Adapathr and ending at Calendar. Two examples of superfluous ‘With’ statements to the same object from package spec and body may be observed. The backward trace constructed from Type_Definitions shows the packages that Type_Definitions is ‘Withed By’. A Graph View is also shown displaying the single cluster in this system. The Parseables body has been relocated manually from its default position next to the Parseables specification to show the dependencies more clearly.

Figure 6. Example Bad Cluster. The figure displays a hypothetical cluster of five packages, characterised by cyclic dependencies between objects. More than one cycle is present in this cluster. Cluster analysis identifies the largest dependency cycle as a cluster.

Subsystem view

SEE-Ada provides a higher level abstraction, the Subsystem View to incorporate design information and manage the complexity of large systems. The Subsystem View does not correspond to a part of the Ada language but Subsystem nodes encapsulate multiple Ada objects.

Figure 7 depicts the Subsystem View of the Ada system shown in Figure 5. The Subsystem View is not generated automatically but is either read in from a file (design output) or constructed manually by allocation of Ada objects to the leaf nodes (shown in double outline) of the Subsystem tree. These nodes may represent Ada libraries, Rational subsystems or other design categories such as DOD-STD-2167A, CSC1/CSC/CSU entities. Figure 7 shows the Profiler subsystem selected and a Layers View of the Profiler subsystem indicating the dependency relationships within the Profiler subsystem itself. Further SEE-Ada features depicted are the
Contains View of the *Clocks* package body (selected), showing the subprograms within the package body and the Source window displaying the source code for the selected *Create_Clocks* subprogram.

**AdaMAT** metrics analysis

One of the motivations for SEE-Ada development was to provide the capability of analysing the large quantity of AdaMAT data. The most recent version of AdaMAT, AdaMAT/D analyses in excess of 300 individual measures (e.g. number of while loops, number of user defined exceptions raised, number of abort statements) which are accumulated on a per package or per subprogram basis, as appropriate. AdaMAT/D also provides semantic based information. These measures are consolidated by a weighting scheme into ten Level 2 Software Quality Criteria (e.g. Anomaly Management, Simplicity, Exactness, System Clarity, Modularity, Self Descriptiveness and Independence), and three Level 1 Software Quality Goals (Reliability, Maintainability and Portability). While the high level metrics require interpretation, low level metrics have direct applicability.

The execution of AdaMAT/D on target code may be slow, taking many hours if all metric acquisition options are turned on. Typically, AdaMAT/D is run in advance of SEE-Ada analysis, the AdaMAT/D output files converted using the SEE-Ada supplied AdaMAT/D conversion tool, and the converted AdaMAT/D output imported into SEE-Ada.

Any of the more than 300 AdaMAT metric items and semantic information may be accessed and displayed on any of the SEE-Ada views. As an example, Figure 8 displays the total *USE_CLAUSES* measure from AdaMAT/D. In the *USE_CLAUSES* Threshold window the user has set ranges and colours for **Use** clause frequency. The Ada packages in the Layers View are each coloured according to the specified frequency ranges. Use clauses are proscribed by many coding standards since, *inter alia*, they render subprograms directly...
Figure 8. Display of AdaMAT USE_CLAUSE Data. The figure shows a Layers View of the Program_Metrics system (Figures 5, 7) with the AdaMAT metric, USE_CLAUSES. The USE_CLAUSES Threshold window allows selected objects to be coloured according to user selected colours and ranges. The 'Use' clause count per object in the example is as follows: white 0, grey 1-5, black 6 or greater. The Show Values Window displays the actual metric value against each Ada object.

visible without the need of the dot notation to indicate the declaring package. In the example in Figure 8, high counts are found for packages coloured black. The Show Values Window lists each package and the Use clause count for each package. The analyst would need to examine the source code of objects with high counts to determine the extent of the problem, but the high values revealed suggest potential problems.

3 SEE-Ada in Software Quality Assessment

Independent validation and verification

A major reason for the development of SEE-Ada was the perceived need for automated support in the Independent Validation and Verification of Ada code. The SEE-Ada views allow analysis of static Ada structure. Dependency analysis, discussed previously (Section 2), provides a significant amount of information on aspects of design and coding quality quickly. SEE-Ada, in conjunction with the code quality metrics generated by tools such as AdaMAT/D, provides a means to readily navigate the large amount of metric data. Metric data from different tools can be loaded into SEE-Ada simultaneously. The analyst would typically select pertinent metrics, examine the distribution of these scores in a Subsystem View, and on finding Subsystems with questionable metric scores, examine the scores of the Ada objects making up that Subsystem using a Layers View. The final stage of analysis would involve inspecting the code of the offending object to determine if a real problem exists.

The following list of problems have been found by DSTO Software Engineering Group in systems examined to date:

- Poor correlation between code and design documents.
- Residual code and null procedures.
- Overuse of sub-units.
- Suppression of Ada Runtime checks.
- Uncommented or poorly commented sections.
— Misuse of Ada features, e.g. generics, use clauses.
— Complex dependencies.

Software development
SEE-Ada may be used to assist the software development process, although it was not specifically designed for this purpose and has not as yet been significantly used for this purpose. SEE-Ada provides information that can be used to identify emerging trends and problems. A system is amenable to analysis as soon as the package specifications are coded. Subsystem and Layers Views allow an initial dependency analysis to be undertaken and the Ada structure to be revised if necessary. During subsequent builds emerging dependency relationships caused by the inclusion of package bodies are revealed. Dependency clusters will not appear until bodies are coded, since cyclic dependencies among Ada package specifications are not possible. This level of analysis can be performed without recourse to AdaMAT metrics and is, therefore, comparatively fast. Iterative development techniques with design, code, test, review cycles provide further opportunity for the effective use of SEE-Ada’s dependency analysis capability. The value of early identification of dependency problems should be self-evident.

SEE-Ada has been used in this manner during the development of SEE-Ada Version 2. A number of clusters emerged during development, and in each case the cause was examined and most of the detected clusters were found to be unnecessary and were eliminated or reduced substantially.

SEE-Ada may be used as part of the review process to show adherence to design decisions, and adherence to coding quality standards. For example, SEE-Ada and AdaMAT/D can detect Ada tasks not specified in the design. As a further example, a tool like AdaMAT/D can count comments and analyse their distribution, and this data loaded into SEE-Ada allows identification of packages that are sub-standard with respect to commenting. While AdaMAT/D takes considerable time to execute and cannot be used indiscriminately, this effort may be feasible for a major review and is more efficient than manual techniques.

4 DISCUSSION
Software visualisation
Modern software systems have become very large and the difficulty of understanding their structure increases with code size. The SEE-Ada Virtual Layers provide an effective abstraction showing a large amount of information in compact graphical form.

A further problem of scale arises in relating large amounts of software metric data to source code. SEE-Ada uses the technique of colouring by statistic to present this metric data in a comprehensible manner. The techniques of reduced representation and colouring by statistic have also been described for the Seesoft C code analysis tool (Eick, Steffen and Sumner, 1992).

Tool integration
SEE-Ada allows integration with other tools via a special data file format. If the output of a tool can be converted to the SEE-Ada standard file format, the output of the tool can be loaded into SEE-Ada. Each analysis tool must be run stand-alone and this is essential for a tool like AdaMAT/D, which may take a long time to execute. SEE-Ada provides integration with AdaMAT/D by way of a special file conversion utility.

It is possible to build special purpose tools to generate data, such as SLOC count, executable statement count, configuration information, comment counts etc. This data can be converted to SEE-Ada standard format and imported. Data from more than one tool can be accessed and displayed simultaneously, once it has been loaded into SEE-Ada. The value of metric data is increased by the rapid access mechanism provided by the graphical user interface.

Capacity
SEE-Ada has been designed to handle the size of current Ada systems under development. The Layers View, in conjunction with the Subsystem View, allows decomposition of large systems into manageable chunks. SEE-Ada has been run successfully on a medium sized system of approximately 1,000 objects (200,000 SLOC, 500,000 total lines of code), but as yet there has been no opportunity to test its capacity on a very large system.

The expected capacity problem with large systems would be the execution time (days) for AdaMAT/D, rather than the capacity of the SEE-Ada tool itself. Reduction in the number of AdaMAT/D metrics accumulated may be a partial solution.

Ada code quality assessment
A number of Australian and overseas defence facilities and development establishments are acquiring SEE-Ada. It is becoming apparent that Ada code delivered to Defence customers will be analysed with SEE-Ada and AdaMAT/D, irrespective of whether such analysis criteria form part of the contractual acceptance criteria. The conclusions of such analysis will form part of the customer’s overall assessment of the quality of the delivered Ada system and will influence the customer’s assessment of the contractor’s capability.

The Department of Defence is already developing a picture of contractor quality capability via the routine analysis of delivered systems by the DSTO Software Engineering Group. The actual end customers will begin to perform this analysis for themselves. It is likely that the quality of delivered Ada code will become increasingly important in the future and the role of Ada quality metrics, such as those produced by AdaMAT/D, will consequently increase.

A particular risk exists that a lack of experience in interpretation of the AdaMAT/D metrics could lead to erroneous
judgements of code quality (Vernik et al, 1991). While the low level metrics of AdaMAT/D are uncontroversial, the high level metrics require interpretation. Some high level measures may be misleading, for example, system dependencies may be specifically located in a small set of packages, consistent with good programming practice, but such a system would contain a few packages with low scores on the Portability index. The AdaMAT/D metrics cannot be used as a simple pass/fail test; however, the metrics can be used to good effect for quickly identifying 'hot spots' for more detailed analysis.

5 CONCLUSIONS
SEE-Ada provides an effective way to analyse Ada dependencies. When used during system development, complex dependency relationships can be identified early and fixed quickly. Early detection of emerging dependency problems should assist in reducing the compilation problems that can occur in large Ada systems.

SEE-Ada used in conjunction with metrics tools, such as AdaMAT/D, provides a powerful validation and verification tool that, in the hands of an experienced analyst, can relatively quickly generate information on the structure and quality of an Ada system.

ACKNOWLEDGMENTS
The authors gratefully acknowledge the assistance of Stefan Landherr, Rudi Vernik and Richard Altmann of Software Engineering Group, Information Technology Division, DSTO in the preparation of this manuscript.

REFERENCES

Vernik, R.J., Turner, I., Baker, C. and Landherr, S.J. (1991): Automated support for assessment of large Ada systems, Proceedings of TRI-Ada '91, pp. 237, New York, USA, The Association for Computer Machinery. (This paper was distributed separately from the proceedings and may be obtained from the authors at Information Technology Division, DSTO, PO Box 1500, Salisbury, South Australia 5108 Australia).

BIOGRAPHICAL NOTE
Grant Slade gained a Bachelor of Applied Science in Computer Studies in 1989 from the South Australian Institute of Technology. He has been with CSC Australia since graduating, and during that time has worked on several large Ada software development programs including the combat system for the Australian Collins class submarine, and the Optus Mobilesat system. Grant worked on the development of SEE-Ada Version 2 as a consultant programmer.

Neville Parker graduated with a Post Graduate Diploma in Computer Science from The University of Adelaide in 1986 and completed an Honours Degree in Computer Science from Flinders University of South Australia in 1989. He worked as a Research Programmer in the Department of Computer Science, University of Adelaide, after which he joined CSC Australia in 1991 where he has managed a number of software development projects and performed a number of consultancy tasks in software engineering and trusted systems. He recently joined Communications Division, DSTO Salisbury as a Senior Professional Officer.
Generic Re-Engineering Environment Design Criteria: An Evaluation of the Software Refinery™

Paul A Bailes
Murray Chapman
Mae Chia
Ian Peake
Language Design Laboratory,
Key Centre for Software Technology,
Department of Computer Science,
The University of Queensland, Qld 4072

1 INTRODUCTION
The rich literature on software re-engineering, e.g. Arnold (1989), McClure (1992), Sneed (1991), Ulrich (1990), suffers from one notable imperfection: a concentration on the (first-order) user functionality of the tools, rather than on the (second-order/meta-level) technical infrastructure from which the quality of the tools ultimately derives. Re-engineering tool users rightly have an interest in this infrastructure, in the same way that applications software users have an interest in the quality of the tools and methods of applications software construction. A clever example is in respect of the maintainability of the applications: users require reassurance that the evolution of their applications is not hampered by poor infrastructure; and so do re-engineers deserve reassurance that meta-maintainability — the maintainability of their metaprogramming trade tools — is preserved.

Admittedly, when commercial imperatives require these tools to be presented as black boxes, there is little opportunity for such consumer discernment. However, when specific re-engineering tools are presented as instantiations of generic re-engineering workbenches or environments, it becomes possible to make estimates about the abovementioned extended tool-quality concerns by assessing the (meta-level) quality of their defining environments.

Our purposes here are therefore:
— to articulate, beyond the first-order user quality issues, the essential second-order infrastructure quality criteria for the design of re-engineering environments. These emerge as a combination of a high-level (meta-)programming toolset along with a relatively persistent object program repository/knowledge base.
— to emphasise the importance of these criteria by in-depth analysis of a leading software re-engineering environment. Reasoning Systems Inc’s Software Refinery™ (also sometimes known by its programming language component “Refine™”) satisfies the above quality criteria, but the difficulties of interfacing it to other tools are acknowledged.
— to propose, on the basis of the analysis, a consequent re-engineering environment design guideline set. The aim is to improve the interworking capabilities of environments such as Refine (and by the way their support for multiple very-high-level programming paradigms) by discovering open repository implementations and facilitating the construction of different (meta-)language interfaces to them.

2 THE PECULIAR NATURE OF RE-ENGINEERING
What makes re-engineering tools special?

2.1 The scope of re-engineering: generalised metaprogramming
While authorities (Arnold, 1992) (Chikofsky and Cross, 1990) differ on vocabulary, re-engineering environments can be expected to support the development of tools to perform the following functions:
— code improvement or restructuring, i.e., eliminating coding standards violations by using better constructs, e.g., replacement of *goto* with "structured" branches and loops;
— code enhancement, i.e., introduction of some new programming paradigm and consequent code restructuring, e.g., object-orientation;
— code conversion, i.e., translation from an old language and environment with limited maintenance prospects to a more modern one, e.g., COBOL and hierarchical databases to 4GLs and relational databases;
— design recovery, i.e., extraction of high-level specification and design information from implementation-level code:
— re-documentation, i.e., synthesis of comments etc. in un- or poorly-documented code.

There seems little point in distinguishing between these terms: in the first place, there are many overlaps. For example, any distinction between code enhancement and design recovery begs the question of what is "implementation" and what is "design". Another example is the effective indistinguishability of code improvement and code enhancement, any distinction depending upon the observer's subjective view of how novel the "better" constructs are.

More useful to us is recognising the essential conceptual unity of these various re-engineering themes. We posit that the single concept of "transition" — inter-language translation (most apparent in code conversion) — actually pervades them all, as follows. Transition is a kind of metaprogramming, in that it either (in fact, both) takes a program ("Origin") as input and generates a program ("Target") as output. If we generalise the concept of target program to include partial, informal or incomplete information, then the other re-engineering activities obviously fall into the ambit of transition.

### 2.2 The difficulty of re-engineering

What distinguishes transition from other kinds of metaprogramming (e.g. compilation), and what consequently distinguishes transition tools from other well-known and widely-available kinds of metaprograms (e.g. compilers)?

#### 2.2.1 Low cost ration: development/use

An initial naive expectation might be that all metaprogamns are not as subject to performance constraints as application programs: the application is executed repeatedly over an extended duration, whereas the relevant corresponding metaprogram (e.g. the compiler) is used only during development. However, fuller awareness of the realities of the software development process (Pressman, 1982) invalidates such a superficial distinction. The development process is itself lengthy, during which the compiler will be used to a degree corresponding to the eventual application, so that performance liabilities in the compiler will significantly detract from productivity. Furthermore, ongoing maintenance throughout the lifetime of the application prolongs the dependence on compiler performance. We are led, therefore, to conclude that an important class of metaprogams, notably compilers, is, in economic terms, indistinguishable from an application program. What different economic factors apply in transition that might justify specialised implementation technology?

The simple answer is that transition is more likely a "one-off" exercise in each case:
— the code to be transitioned already works, i.e. the elimination of requirements anomalies or programming errors that require repeated use of the compiler are not a concern in transition;
— the original code input to the transition can be eventually discarded at the end of the transition process, unlike development source code which must be retained and re-compiled during maintenance.

This implies that, in the facilitation of the transition process, rapid tool development is the dominant factor, far outweighing the performance (as opposed to the functional) characteristics of the transition tool. This is a familiar situation in software engineering, echoing the technical rationale for special-purpose Rapid Prototyping (Budd et al., 1984) tools, where it is also the case that development dominates the use of tools. In other words, very-high-level languages should be used for transition metaprogramming.

#### 2.2.2 Idiomatic requirements

Another difference between transition and compilation is that compilers operate in circumstances that are both *simple* and *objective*. Simple, because the target code produced by a compiler exists only to be executed: there are no stylistic criteria applicable to the quality of the output. Objective, because the analysis of origin code and the complementary translation to the target are specified entirely independently of particular origin programs.

By way of contrast for transition, simplicity is violated by the requirement that target code then becomes the maintainable source code, and the resulting complexities violate objectivity as follows.

#### 2.2.3 General multiplicity of possible representations

Most *programming* languages possess universal expressive power, and automatic translation between languages of equivalent expressive power is relatively straightforward. However, these objective translations can involve quite arbitrary transformations on the structure of the origin code. For example, in the extreme case, the origin code can be transformed into a data structure of the target language, and executed by an interpreter for the origin written in the target. This is an impractical solution for transition because of the following direct and indirect reasons:
— directly, interpretation reduces execution speed (in this case, the transitioned application) by an order of magnitude;
— directly, concrete environment specifics of the origin (as opposed to abstract computational properties) that necessarily involve performance issues (e.g. IO interfacing) may not be able to be interpreted;
— indirectly, construction of the interpreter is a significant problem in itself.

The underlying problem, exquisitely revealed by the example, is that the one concept may be expressed in a multiplicity of ways within a single programming language. Starting with the “natural” expression of some algorithm or data structure, we can move to the degenerate cases of interpretation of another language and its implementation via a wide range of successively more contorted renditions. The question is one of how to map the way in which a concept is rendered in the target language into the best rendition in the target language out of many choices of target language rendition.

2.2.4 Specific origin-target incompatibilities

Now, the severity of the problem should not be exaggerated. For languages of similar structures, simple localised translations that choose the correct target solution are trivial. However, when use is made of an origin structure that has no target counterpart, or vice versa, the problem reappears. Our solution grants special status to selected constructions — idioms — in the origin that we do not expect to be able to simply translate automatically. The bases for their selection are as follows.

— **Origin-relative-to-target deficiencies**: when the target is in some sense less expressive than the origin. These may either be rather absolute, in the case of unusual languages like Prolog; or rather relative to the intended target, in the case where both origin and target are familiar languages but that retain important distinctions, e.g. translating C recursive functions into COBOL.

— **Target-relative-to-origin deficiencies**: when the origin is unable to express its function directly in a way that would be easy in the target. This is the classic re-engineering example, embracing the abovementioned activities of code enhancement, conversion, design recovery and even re-documentation.

— **Environmental idiosyncrasies**, such as data models, screen formats and I/O processing, in combinations corresponding to the above.

The problem common to each of these is to identify what is “really happening” in the origin program, and to find the natural corresponding expression in the target.

2.2.5 Conclusions

Automatic identification of idioms is difficult (emphasising the necessity of high-level metaprogramming tools — see above) and sometimes impossible. Though constrained to be expressed in terms of particular (programming) languages, idioms have much in common with the constructs of natural languages in that they are unformalised human creations out of some ultimately restricted design space. Idioms are created out of programming languages, and by comparison natural languages are created out of a complex of abstract concepts and concrete symbols for their representations. The foundations of mathematics and computability theory show us that these design spaces correspond. A programming language is a formalisation of constructive mathematical logic, but within that space there is effectively as much scope for human creativity as in the development and use of natural language.

We therefore conclude that the recognition and analysis of idioms is as difficult as natural language understanding. As this remains a largely open research problem, nobody can hope to perform idiom analysis perfectly. Rather, transition tools must make provision for the application of human problem-solving skills as an integral part of the process, but in such a way as not to detract from the full automation of the overwhelming majority of transition effort. Satisfaction of this goal is the major constraint on the architecture, and on the implementation technology. Notably, this constraint is characteristic of at least one important trend in so-called Knowledge-Based Software Engineering (KBSE) (Green et al 1983), to the results of which we look for solutions to our problem.

3 TECHNOLOGICAL DEMANDS OF RE-ENGINEERING

In addition to the obvious necessity of general metaprogramming capabilities (syntax analysis, abstract syntax tree construction and manipulation), what technological requirements do the above special circumstances induce?

3.1 Human intervention

At the abstract level, the issue of human intervention is not a problem — even when using very-high-level meta-languages, we can expect a significant degree of human intervention in instantiating them with the details of particular origin language and idioms and target language details. (Not that we resile from attempting to simplify and automate these processes as much as possible — see below). The critical problem is how to reconcile the desire for the full-automation of the concrete level of transition with the necessity of retaining at least the possibility of human intervention.

Contemporary compiler technology provides a negative illustration. Whilst fully automatic, from syntax analysis through to code generation, a compiler provides no realistic opportunity for a human operator to review and alter its decisions. *Post hoc* intervention — reviewing compiler output for satisfactory performance — is a universal, but degenerate possibility, failing to provide any constructive assistance with any remedial process. Debugging tools and environments in principle allow a compiler to admit some *inter alia* intervention, but are generally so far removed from the terminology of the application (in the compiler/transition case, of the origin and target languages) to negate their practical utility.

The implication is that the abstract, internal representation of origin or target source should be amenable to interactive access by a human operator, using high-level browsing and manipulation tools. In other words, a persistent repository should be used for representing abstract syntax.

---

**THE AUSTRALIAN COMPUTER JOURNAL, VOL. 26, No 4, NOVEMBER 1994** 153
3.2 Linguistic expressiveness
Aside from the general very-high-level requirement, the following special meta-linguistic requirements prevail.

3.2.1 Extensibility
At the concrete-transition level, the existence of idioms implies that whatever basic components of transition tools for specific origin and target languages are developed will vary from instance to instance based either on newly-discovered differences between origin and target languages that are combined for the first time, or on new environmental circumstances. Re-use implies that these changes be accommodated by adapting existing tools and components, or in other words, by extending generic core components.

At the abstract transition tool level, we first contemplate the requirements placed upon the eventual transition base technology. Their summation is the ability to support ill-defined metaprogramming tasks, simultaneously minimising development effort through maximised expressiveness.

3.2.2 Knowledge-based
An important KBSE theme is the "transformational paradigm" (Kotik and Markosian, 1990): application of powerful rewrite rules to a problem formulation in order to generate a logically-equivalent statement but in an acceptable format (the answer). The paradigm superficially has much in common with classic AI/theorem-proving approaches, e.g. the use of very-high-level logic-style languages for specifying the transformations, but there is one important difference — the presentation of the VHLL as an interactive expression evaluator with operator access to intermediate data structures. Even this limited form of persistence suffices to allow an impressive amount of human assistance to otherwise limited automatic tools.

4 RE-ENGINEERING WITH SOFTWARE REFINERY

4.1 Overview
We identify the Software Refinery™ metaprogramming environment and its embedded Refine™ language as meeting all the above criteria, as follows:

— The Refine language is a wide-spectrum presentation of a multiplicity of modern high-level programming paradigms: structured procedural; applicative; functional; logic; object-oriented; compiler-compiler/syntax-pattern-directed. This guarantees the maximum possible degree of programmer expressiveness, as peculiarly required by transition.

— Refine is extensible in two dimensions: it possesses ad-equate modular structure, so that incorporation into a generic tool of origin-target combination specifics can be structured; and extended syntactic forms, such as may be used to describe origin/target language forms, may be directly embedded in Refine source.

— The Software Refinery is presented as an interactive expression evaluator over a proprietary persistent object repository. A predefined library provides powerful operations for program development. Large programs may be developed incrementally, their components being tested within the development tool. User routines may be used to extend the system library. Most importantly, data structures constructed during Refine executions (the "object base") are available as quasi-persistent, global data to the command interpreter, the system library providing high-level, graphical tools for the exploration and alteration of which. In particular, the internal stages of a origin-target transformation become amenable to human operator alteration.

Indeed, the Software Refinery represents the archetypal embodiment of the transformational paradigm, having been used to develop the impressive KIDS tool (Smith, 1990) for the refinement of formal specifications into executable programs, in addition to the re-engineering applications cited above (Kotik and Markosian, 1990).

Moreover, the Refine environment is well-integrated with the generic X11 and Emacs environments, implying that sophisticated graphic-, window- and text-management functions are immediately available.

4.2 A generic re-engineering architecture
From these general capabilities, we have drawn out a specific combination of functions in a generic architecture for transition (Bailes, Chilvers and Peake, 1993), summarised as a sequential process as follows, and as illustrated by the following figure.

![Semi-Automated Transition Architecture](image)

Figure 1. Semi-Automated Transition Architecture (abbreviations as defined below).

4.2.1 SA — syntactic analysis
The first phase of transition is the construction of a manipulable internal representation — fully automatic with Refine.

4.2.2 CN — censoring
The next phase is to elide origin components that can’t be processed automatically. The options are

DG dodging — marking abstract tree components as invisible to automatic translation TR (below)

DL deletion — removing abstract tree components that are irrelevant to the translation.

4.2.3 TR — transformation
Replacing origin constructs with target constructs involves

RO re-orientation — replacing origin program macro-level structures with target structures

TN translation — replacing origin program minutae with target minutae.
Note that CN and TR take place iteratively, so that both the following facilities are open to the human operator:
— the choice of applying partial transformations to the tree, with further transformations dependent upon inspecting the outcome;
— applying partial transformations similarly partially, with the partial transformations selected to avoid improperly censored sub-trees.

4.2.4 DS — differential synthesis
Generation of target program source is complicated by the presence in the abstract tree of dodged origin source that remains untranslated, and so the synthesised code must indicate when origin, not target code, is in place.

4.2.5 RC — reconciliation
Finally, dodged origin source represents material that is important to the translation, but which is not translated using the internal representation (typically because even automatic partial translation is not cost-effective). The human operator identifies this residue (from DS above) and manually translates it.

5 EXPERIENCE
Our practical experience with Refine complements the theoretical benefits outlined above. Our notable engagements have concerned development of an extension to Refine itself for dynamic sub-typing (Bailes, Chapman, Gong and Peake, 1992), and an industrial transition case study to translate COBOL (with embedded hierarchical data base operations) into corresponding Ingres (operating on a relational data base). The essential technical challenge was re-orienting into the Ingres field-activation-block structures.

Some specific observations are as follows.

5.1 Notable benefits

5.1.1 Involution
Refine’s self-implementation simplifies metaprogramming in that the same mechanisms are used for the manipulation of the programs being processed and of the metaprograms that process them. For example, metaprogram debugging has the same flavour as censoring.

5.1.2 Automatic pretty-printing
An important consequence of the combination of automatic pretty-printing of source code from internal abstract trees and involution is that Refine debug information is automatically displayed in source form.

5.1.3 Incremental compilation
Incremental compilation of procedures was essential, lest the time spent in module compilation and linking significantly retard the development process. Incidentally, incremental compilation provides much of the benefit with little of the cost and inflexibility of language-specific editors.

5.1.4 Adequate polymorphism
The type system is not so strong as to preclude polymorphic functions and data structures. It is a moot point (Goodwin, 1981) as to whether or not a stronger, static polymorphic typing system would be appropriate for this application.

5.1.5 Powerful tree operations
Programming phase TR was greatly simplified by the combination of: Refine transforms; their formation into rule structures; and the controlled application of these by Refine’s canned traversals.

5.1.6 Coupling of concrete and abstract syntax
At first glance, Refine’s presentation of language models is confusing, e.g. in that right hand sides of production rules do not appear to make reference to (other) non-terminals. However, this is more than compensated for by there being no need to explicitly transform the (implicit) concrete tree into the desired abstract tree (i.e., domain model representation) as would be required in conventional compiler-compilers.

5.1.7 Multi-lingual domain models
An interesting consequence of the combination of automatic pretty-printing and the polymorphism of the Refine object base is that not only can the hybrid domains resulting from RO and TN be represented, but also can be displayed in DS with their respective concrete syntax allegiances clearly displayed. Indeed, DS is implemented trivially by Refine.

5.2 Negative technical minutae
A number of detailed problems have arisen that will be referred to Reasoning Systems Inc for possible rectification in further releases of Refine.

5.2.1 Inflexible tree attributes
Two important operations on internal abstract tree structures are: comparison — the basis for pattern matching; and copying. Conceptually, copying can be either “deep” (copying all elements in a referenced structure) or “shallow” (copying only the pointer to the root). Comparison can be based upon essential tree nodes or alternatively take account of decorations. Combinations of these possibilities are all useful in transition, but Refine supports only deep copying combined with essential tree node comparison. The other three have to be programmed explicitly.

5.2.2 Non-incremental grammar compilation
One of our research goals is to replace Dialect (the Refine language modelling sub-system) with a more powerful syntax analyser. In the meantime however, incremental compilation of the existing grammar formalism would greatly hasten development efforts using Refine.

5.2.3 Uncontrolled pretty-printing
When mapping back into concrete from abstract internal form (as in DS), any formatting different to that provided by Refine
has to be programmed explicitly in entirety. Selective over-riding of Refine defaults should be supported, especially for transition, e.g. to admit site-specific coding conventions.

6 AN OPEN MANDATE FOR RE-ENGINEERING ENVIRONMENT DESIGN

Despite its strengths, Refine has the drawback of failing to integrate well, or effectively even at all, with other metaprogramming languages. For example, it may be desirable in some application to bring to bear the full expressive power of Prolog on some especially-difficult rule-based transformation procedure, but there is no provision e.g. for Refine programs to invoke Prolog subroutines. There is in fact a C linkage facility, but a more general solution is necessary.

6.1 Advantages of an open environment for re-engineering

A further problem is that the Refine object base is persistent only for the duration of an interactive session, and that it therefore must be explicitly saved for later use. This might be overcome with automatic load and save facilities, a trivial extension to the Software Refinery indeed, but a further problem with this persistence model is that shared access to the object base by multiple users is a priori excluded. This might not be so much of a problem when re-engineering small source files, but could be a difficulty when more than one human operator is sharing the task of transforming a large single source file. At the very least, we do not want our technology to limit our operational capabilities.

Both these problems can be solved by using an open, language-independent representation for the persistent object base. Language independence means that it should be possible to make bindings from arbitrary languages to the object base, so that different re-engineering functions can be programmed in the most appropriate languages. Persistence further requires that this representation be defined at the operating/file system level, which then raises the possibility of fine-grained access controls to the persistent information.

6.2 Towards a generic open environment for re-engineering (GOER)

The desired open re-engineering environment possesses the characteristics forming a logical progression as follows:

(P) Persistent Storage
(U) Multi-User Shared Access

6.2.1 Persistent storage
Developing the Refine object base into a persistent system addresses two issues:

— a disk-based representation of program representations can conceivably employ operating-system-level mechanisms to manage multi-user access;

— our promulgation of such a representation then provides an open standard to which bindings for multiple languages can be targeted.

Therefore, the prime conceptual step is

(P1) Specify and implement a disk representation for the Refine object base — our "Open Object Store" (OOS).

Once this is achieved, the tasks of binding other languages to the OOS can be decoupled from implementing the OOS to allow multi-users.

6.2.2 Multi-language access

The abstract behaviour of the OOS is determined by the data types and operations thereon that can be dealt with by client programs. The language in which these types are defined forms a kind of data definition language (DDL). Bindings to the OOS for different languages involve the construction of language-specific interfaces to the OOS data abstractions.

(L1) Bind Refine — OOS. Corresponding to the standardisation of an external representation for Refine objects. Refine object type declarations, and operations thereon, will have to be compiled into the basic operations on the new representation.

(L2) Bind C — OOS. The first test of multi-language accessibility is to provide a C binding to our persistent object base. This will involve providing an appropriate DDL for C, akin to Refine’s object type hierarchy, and presumably similarly designed and implemented.

(L3) Adapt C metaprogramming tools. We will adapt available C metaprogramming tools (e.g. as made available by the OSF/GNU efforts) to use the OOS, as manipulated via the DDL.

(L4) Finalise open object/data definition language design. The hosting of the Refine DDL in C will provoke a comparison between the object-oriented functionality of Refine, C, and “true” object-oriented languages such as C++ and Modula-3. This will hopefully lead to the adoption of one existing language's model of object orientation for our ultimate Open Object Definition Language (OODL).

(L5) Revise bindings. Stages L1-L3 will have to be rereacted to accommodate the new OODL bindings.

Subsequent activity will include:

— re-engineering existing Refine tools to use the new Refine/ OODL bindings;

— adding bindings for other languages.

6.2.3 Multi-user shared access

(U1) The initial OOS (P1 above) will investigate the UNIX filing system (UFS) as a representation for the OOS. Advantages of the UFS for this purpose include: the hierarchical, linked structure is immediately useful for representing abstract syntax graphs; there already exist UFS tools to supply much of the functionality required for a generic re-engineering environment (e.g. directory browsers, even basic commands to make and break links); multi-use is automatically available.
(U2) Investigate the efficiency of alternative OOS representations. The big argument against a UFS-based representation is that it has slow response. We aim to demonstrate the veracity of this complaint, by comparison with OOS representations based on custom-built tree-structured files, and on some available Object-Oriented Data Bases (e.g. ONTOS, Aditi).

(U3) Reconcile with linguistic developments. As the OODL evolves (L1, L2, ...), so will the behavioural requirements of the OOS. This final stage involves choosing one of the alternative multi-user OOS technologies, and implementing therein the concepts of the ultimate OODL (L4 above).

7 CONCLUSIONS
Our first-order result is to assess that the Software Refinery is broadly satisfactory as a generic re-engineering tool development environment, but with some notable avenues of improvement.

Our second-order result is to have articulated principles for the assessment of other tools such as Software Refinery. However, while such competitors are at present hard to identify, the principles can immediately be used in the alternative role of inspiring the necessary identified enhancements to Software Refinery.

ACKNOWLEDGEMENTS
Our use of the Software Refinery is supported by the Australian Defence Science and Technology Organisation (DSTO) through its Australian Software Engineering Environments Consortium (ASEEC), and our commercial application was carried out under the auspices of the Centre for Information Technology Research (CITR) of The University of Queensland.

“Software Refinery” and “Refine” are trade marks of Reasoning Systems Inc.

REFERENCES

BIOGRAPHICAL NOTES
Paul Bailes is Professor of Computer Science at The University of Queensland, Brisbane, Australia where he leads a research programme in language design, language processing and software re-engineering within the Key Centre for Software Technology. He earned a BSc (Hons.) in 1978, and a PhD in 1984, both in the field of Computer Science at The University of Queensland, and has had additional teaching and research experience at Wollongong, Griffith and Kent Universities.

Murray Chapman is a Senior Research Assistant within the Key Centre for Software Technology, Department of Computer Science, The University of Queensland, from which he graduated BSc (Hons.) in 1984, both in the field of Computer Science at The University of Queensland, and has had additional teaching and research experience at Wollongong, Griffith and Kent Universities.

Mae Chia holds BSc, DipInfTech and MinfTechStud qualifications, earned in the Department of Computer Science, The University of Queensland. She is currently working in Singapore.

Ian Peake is a Research Scholar within the Key Centre for Software Technology, Department of Computer Science, The University of Queensland, from which he graduated BSc (Hons.) in Computer Science in 1991. His PhD research investigates language extension as a software reengineering tool development paradigm.
Graph-based Class Testing

Daniel Hoffman
Department of Computer Science,
University of Victoria,
PO Box 3055, Victoria, BC, V8W 3P6 Canada

Paul Strooper
Department of Computer Science,
University of Queensland,
St Lucia, Queensland 4072 Australia

In contrast to the explosion of activity in object-oriented design and programming, little attention has been given to object testing. Two tasks arise: (1) testing a base class \( C \) and (2) testing a class derived from \( C \), assuming \( C \) has been thoroughly tested. We describe test inputs as traces: sequences of function calls. Testgraphs are used to specify the large number of traces typically required for thorough testing. A testgraph partially models the states and transitions of a class implementation. Testgraph traversal may be varied to produce different test inputs. Outputs are verified using an executable oracle: a class specifically developed to check output correctness. Surprisingly thorough testing is achievable with simple testgraphs and oracles. The key is designing the two together, to avoid inputs for which output checking is hard. In testing derived classes, we focus on reducing the cost of test development by reusing base class tests and test code. Testgraphs provide a convenient structure for such reuse. We illustrate our approach using detailed examples.

Keywords: testing, automated, object orientation, class, reuse, inheritance.

1 INTRODUCTION

In the object-oriented dream, productivity and reliability are vastly improved, primarily through reuse. Because reliability depends critically on testing, this dream cannot become a reality without effective class testing. However, even though testing is standard practice in industry (Beizer, 1984; Myers, 1979), little has been published about the testing of object-oriented software.

The tests for class \( C \) must be repeated many times: for the initial version of \( C \), after each modification to \( C \), and when \( C \) is used in a new environment, e.g., a new operating system or compiler. This requirement for repeatability suggests that test execution should be automated. Automation of test development is less important and also far more difficult.

Inheritance introduces special testing concerns. Suppose that class \( C \) has been thoroughly tested. What testing is appropriate for \( C_D \), derived from \( C \)? Obviously, member functions introduced or redefined in \( C_D \) must be tested. Less obviously, functions inherited from \( C \) may need to be retested (Perry, 1990). Previous work focuses on minimising the retesting, despite the substantial effort required to do so (Harrold, McGregor, and Fitzpatrick, 1992). We choose a different approach. Even if there are no classes derived from \( C \), \( C \)'s tests will have to be repeated often. Thus, instead of avoiding retesting, we expect to do it frequently, and focus on making it as inexpensive as possible. For new or redefined member functions, we reduce the cost of test development for \( C_D \) by reusing \( C \)'s test code.

We focus on collection classes rather than graphical user interface classes. Thus, we avoid the difficult problems of keyboard and mouse capture and playback, and screen capture and comparison, and substantially increase the potential for cost-effective automation.

2 RELATED WORK

In reviewing the literature on object testing, we distinguish the testing of base classes, ignoring inheritance, and the testing of derived classes. Testing base classes is similar to testing software modules, and early work by Panzl (1978) on the regression testing of Fortran subroutines addresses some of the issues. The DAISTS (Gannon, McMullin, and Hamlet, 1981), PGMGEN (Hoffman, 1989), and Protest (Hoffman and Strooper, 1991) systems all automate the testing of modules based on test cases using sequences of calls. While the DAISTS system requires a formal algebraic specification of the module under test, the PGMGEN system uses a simple test script language, and the Protest system uses test cases defined by a Prolog program. Frankl (1990) has developed a scheme for object-oriented testing using algebraic specifications. To test an object \( O \), the specification is used to generate pairs of equivalent call sequences. Two instances of \( O \) are created, and one of the call sequences is executed on each instance. Object behaviour is deemed correct if the two instances are left in the same internal state.
Little work has been done in the area of testing derived classes. Fiedler (1989) describes a case study on testing C++ objects, but provides little information on how derived classes were tested. Perry and Kaiser (1990) show that, in general, functions inherited from a base class may need retesting in the context of the derived class. Harrold, McGregor, and Fitzpatrick (1992) extend this work by considering what member functions must be retested, based on the way in which the class is derived from the base class. Smith and Robson (1992) propose a method and framework for testing object-oriented programs, but no concrete examples of testing derived classes are given. Murphy and Wong (1992) proposed a testing methodology for object-oriented systems. In this framework, the ACE tool (Wong, 1992) was developed to support the testing of Eiffel and C++ classes. ACE is an enhancement of PGMGEN (Hoffman, 1989) that provides a mechanism to override individual test cases for the testing of derived classes. Turner and Robson (1992) propose a state-based approach to object testing, in which, just as for testgraphs, test cases are defined from the possible states of an object.

3 TERMINOLOGY

In this paper, each test suite focuses on a single class: the Class Under Test or CUT. A test oracle is a means for checking the correctness of the outputs computed by the CUT. Oracles take on a variety of forms, including:

- manual examination of each output from each test run
- manual generation of each output once, to be compared automatically to the output from each test run
- programs that generate \((x,y)\) pairs, where \(y\) is the correct output for input \(x\)
- programs that generate the correct output \(y\) for any input \(x\)
- programs that determine the correctness of any input/output pair \((x,y)\).

All testing requires an oracle of some kind; we consider only automated oracles.

We focus on the testing of C++ classes and therefore use the terminology of C++ throughout (Stroustrup, 1991). By convention, in member-function names we use the prefix \(s_\) (set) to indicate calls that set internal class values, \(g_\) (get) to indicate calls that retrieve those values, and \(sg_\) for calls that do both. A trace is a sequence of function calls.

We illustrate these ideas on the intsetO class, providing access to a bounded set of integers. A partial class declaration

```cpp
class intsetO {
public:
    intsetO(int);
    void s_add(int);
    void s_del(int);
    int g_mem(int);
};
```

Figure 1: intsetO — class declaration.

Figure 2 shows the call intsetO(n) instantiates an object with the set initially empty and with maximum size \(n\). s_add(i) adds the integer \(i\) to the set, s_del(i) deletes \(i\) from the set, and g_mem(i) returns true if and only if \(i\) belongs to the set. The call s_add(i) signals the exception MEM if \(i\) already belongs to the set, and FULL if there are \(n\) elements in the set. s_del(i) signals NOTMEM if \(i\) does not belong to the set. We assume that calls that cause exceptions do not change the class state. The trace

```cpp
intsetO(5); s_add(1); s_add(2); g_mem(1)
```

is exception-free; the g_mem call returns true. In the trace

```cpp
intsetO(5); s_add(1); s_add(1)
```

the second s_add call signals the exception MEM. We use a simple exception signalling mechanism: class C signals exception e by invoking a function named e, usually implemented by the user of C. For example, to signal exception MEM, the s_add implementation invokes the function MEM.

In the next section, intsetO is the CUT. Additional classes will be introduced to perform the testing; one of these classes will serve as test oracle.

4 TESTING BASE CLASSES

4.1 Testgraphs

**Definition**

A testgraph is a directed graph with node and arc labels, and a designated start node. A path in testgraph \(T\) is a sequence of nodes \((n_0,n_1,\ldots,n_k)\) where, for every \(i \in [1,k]\), \(T\) contains an arc from \(n_{i-1}\) to \(n_i\). Associated with each path in \(T\) is a path label: the sequence of node and arc labels encountered on the path.

For the testgraph in Figure 2, \((n_0,n_1,n_2)\) is a path with path label \((a,x,b,y,c)\). In pictures of graphs, we use an arc with no source node to indicate the start node \((n_0\) in this case).

![Figure 2: Simple testgraph.](image)

**Tester's interpretation**

A testgraph is a partial model of a CUT. Each testgraph node corresponds to a CUT state, and the start node to the initial state; each arc corresponds to a state transition in the CUT. While the CUT state space is normally very large, the testgraph state space is vastly smaller. Roughly speaking, the CUT state space contains all states that can be reached; the testgraph state space contains those states that will be reached by the test suite.

**Figure 2: Simple testgraph.**
The node label is a trace chosen to determine whether the CUT is in the correct state and whether the CUT operates properly in that state. The label \( x \) on an arc from \( n_s \) to \( n_d \) indicates that, if the CUT is in state \( n_s \), then the calls in trace \( x \) will leave the CUT in state \( n_d \). Consequently, each path corresponds to a sequence of CUT states and transitions. The path label is a trace that will generate the transitions, checking each testgraph state along the way.

We normally adhere to two design rules for testgraphs. Since node labels are used to check whether the CUT is in the correct state, they do not contain calls that alter the CUT state, i.e., they contain only get calls, and calls that signal exceptions. Similarly, arc labels contain only calls that do alter the CUT state, i.e., exception-free set calls.

To add flexibility to the testgraph scheme, we associate a parameter with each testgraph. With this parameter, each testgraph becomes a family of testgraphs, one for each parameter value. By supplying a value for the parameter, the tester selects a particular testgraph to be tested. Note that the testgraph parameter influences the graph labels, but not the graph topology.

Graph coverage

We generate test cases by traversing the testgraph beginning at the start node; test cases are derived from the path labels and a set of path labels constitutes a test suite. We are interested in test suites that, in some sense, cover the testgraph. We consider three types of coverage: node coverage, arc coverage, and path coverage — analogous to statement, branch, and path coverage in structural testing (Howden, 1976). As in structural testing, arc coverage subsumes node coverage, and path coverage is difficult to achieve — impossible if the testgraph is cyclic. At present, we use traversals that achieve arc coverage.

intset0 testgraph

A testgraph for the intset0 class is shown in Figure 3. The node and arc labels shown suggest the required traces, specified in detail in the Implementation section below. The testgraph nodes are chosen to include sets of size 0, \( M \), and 10 \((0 < M < 10)\) with elements that are easy to generate automatically. In Figure 3, beside each node is the CUT set it represents. We have chosen the node label to check that \( g_{\text{mem}}(x) \) reflects the expected set contents, and that \( \text{FULL}, \text{MEM} \) and \( \text{NOTMEM} \) are signalled appropriately. The arc labels are straightforward. For example, for the label \( \text{DELODD} \) we have

\[
\text{s_del}(1); \text{s_del}(3); \ldots; \text{s_del}(9)
\]

Arc coverage is achieved with two paths:

(EMPTY, ALL, EVEN, EMPTY)

and

(EMPTY, ALL, ODD, EMPTY)

The graph is parameterised by a single integer: the parameter to the \textit{intset0} constructor.

4.2 Test oracles

Automated testing produces large numbers of test cases. Manual checking of outputs is typically infeasible, making oracle construction a major concern in test development. To perform the oracle task for CUT \( C \), we introduce the oracle class \( C_0 \), similar to \( C \), except that its states and transitions are restricted to those in the testgraph. \( C_0 \)'s state variables represent the testgraph nodes; its member functions compute the testgraph transitions and help in verifying the CUT behaviour. The oracle is incorporated into the testgraph scheme by adding oracle calls to the node and arc labels. Oracle calls are added to arc labels to keep the oracle state synchronized with the CUT state, and to node labels to evaluate CUT behaviour.

4.3 Implementation

Figures 4 and 5 show partial C++ class declarations for the oracle and testgraph classes. Let \( N \) be the graph parameter: the maximum set size passed to the \textit{intset0} and \textit{intset0\_o} constructors.

```cpp
class intset0_o {
public:
    intset0_o(int);
    void union(int);
    void diff(int);
    int g_mem(int);
    int nod;
    int maxsiz;
};
```

Figure 4: intset0 oracle — class declaration.

In Figure 4, the state variable \( \text{nod} \) stores the current testgraph node as an integer, interpreted as a two-bit string:

Let \( s_o \) be an object of type \textit{intset0\_o}. Viewing the node numbers \( s_o.\text{nod} \) and \( n \) as the integer sets they represent, \( s_o.\text{union}(n) \) assigns \( s_o.\text{nod} \cup n \) to \( s_o.\text{nod} \), \( s_o.\text{diff}(n) \) assigns \( s_o.\text{nod} \cap n \) to \( s_o.\text{nod} \), and \( s_o.g_{\text{mem}}(x) \) returns \( x \in s_o.\text{nod} \). Note
that the parameters to union and diff represent sets, while the parameter to g_mem is a set element. With the bit representation and the C++ bit operators, implementations for union and diff are trivial one-liners; the g_mem implementation is a simple four-way case statement.

In Figure 5, the testgraph nodes and arcs are represented as integer constants. The behaviour of the intset0_testgraph member functions is summarised below. Let s be an object of type intset0 and let s_o be an object of type intset0_o.

- runtests: drives the testgraph traversal by repeated calls to arc and nod.
- arc(s, s_o, a): generates the transition indicated by arc a in both s and s_o.
- nod(s, s_o): checks the normal and exceptional behaviour of s, using the oracle and the remaining four member functions.
- chkmem(s, s_o): checks for differences in s.g_mem(i) and s_o.g_mem(i), for all i ∈ [1, N].
- chkexcfull(s, s_o): if s_o.nod is ALL, invokes s.s_add(-1) and checks that exception FULL is signalled.
- chkexcnotmem(s, s_o): checks that exception NOTMEM is signalled by s.s_del(i) for all i not in s_o.nod.

At present, graph traversal is accomplished by hand-coding runtests with calls to arc and nod; 12 such calls are required to achieve arc coverage for this example. With larger testgraphs, hand-coded traversal is no hardship; with larger graphs automation will be essential and appears feasible.

const int ADDALL = 0;
const int DELODD = 1;
const int DELEVEN = 2;
const int EMPTY = 0;
const int EVEN = 1;
const int ODD = 2;
const int ALL = 3;

class intset0_testgraph {
    public:
        void runtests();
        void arc(intset0 &, intset0_o &, int);
        void nod(intset0 &, intset0_o &);
        void chkmem(intset0 &, intset0_o &);
        void chkexcfull(intset0 &, intset0_o &);
        void chkexcnotmem(intset0 &, intset0_o &);
    }
}

5 TESTING DERIVED CLASSES
To reduce the cost of test development for a derived class, we reuse the base class tests and the test code. This leads to the intuitively appealing notion of a “test hierarchy” that is isomorphic to the production hierarchy. That is, the testing of a derived class is derived from the testing of the parent class. As the examples below illustrate, with testgraphs this can be accomplished by simply redefining the labels of arcs and nodes, or by adding/deleting nodes and arcs to/from the testgraph.

5.1 Example — intset1
We first examine intset1, a trivial extension to intset0, where we have added the member function g_siz, which returns the size of the set. Figure 6 shows the class declaration for intset1. The corresponding extension to the testing is also very simple. We add the call g_siz() to every node label of the testgraph, and we add a call to the oracle to return the size of the sets in the testgraph.

class intset1 : public intset0 {
    public:
        intset1(int);
        int g_siz();
    };

Figure 6: intset1 — class declaration.

To implement this scheme, the oracle and testgraph classes for intset1 are derived from those for intset0. The oracle class is extended by the member function g_siz, which is the oracle version of intset0::g_siz. Since the testgraph contains only four states, g_siz is implemented as a simple four-way case statement. To append g_siz() to every node label in the testgraph class, we define the intset1 version of nod so that it calls the testgraph member function chksize after calling intset0::nod. The function chksize(s, s_o) reports the differences between s.g_siz() and s_o.g_siz(), where s is of type intset1 and s_o of type intset1_o.

Note that the graph traversal for intset1 is the same as for intset0, and that all the tests that are performed for intset0 are automatically repeated for intset1.

5.2 Example — intset2
The second example, intset2, extends intset1 by providing a method to iterate over the elements in the set. This is a very natural extension to a collection class, and a collection class typically provides at least one such iterator. Part of the class declaration for intset2 is shown in Figure 7. An intset2 object operates either in set (SET) or sequential (SEQ) mode, and the enumerated type mod is used to represent these two modes. The
typedef enum SET,SEQ mod;

class intset2:public intset1 {
    public:
        intset2(int);
        void s_add(int);
        void s_del(int);
        void s_mod(mod);
        int sg_next();
        int g_end();
};

Figure 7: intset2 — class declaration.

member function s_mod is used to switch between the modes. In SET mode, intset2 behaves like intset1. In SEQ mode, the user can iterate over the elements of the set with the member functions sg_next and g_end, as follows. Following the call s_mod(SEQ), each successive call to the function sg_next returns a different element of the set, signaling END when there are no more elements to return. There are no restrictions on the order in which the elements are generated, but each element must be returned exactly once. The member function g_end returns true if all elements have been returned by sg_next, and false otherwise. Finally, the calls s_add and s_del are modified to signal the exception SEQMOD when the object is in SEQ mode; similarly, sg_next and g_end signal SETMOD when the object is in set mode.

In this case, the testgraph for intset2 is obtained from the testgraph for intset1 by adding four nodes and eight arcs, as shown in Figure 8. Although we have added four nodes, we have given them the same symbolic name (SEQ) in Figure 8, because they have the same label. This label is a trace that checks the sequence of elements returned by sg_next. Note that to build a general oracle that does a complete check on the elements returned is hard, because sg_next may return the elements in any order. To reduce the cost of building and maintaining the oracle, we choose a “partial” oracle that does convincing cross-checking, but at a substantially lower cost than a full oracle. In particular, our oracle checks (1) that every element returned by sg_next is in the set, (2) that the correct number of elements is returned, and (3) that the sum of all the elements returned is correct. This is a partial oracle, because, for example, an incorrect implementation that returns the number 2 three times when the actual set is \{1, 2, 3\} will be deemed correct by this oracle. However, this oracle is trivial to implement, it handles the nondeterminism of sg_next, and we expect that it will catch most incorrect implementations. To simplify Figure 8, we have omitted labels from the eight new arcs, but their labels are straightforward: the four arcs with destination SEQ have the call s_mod(SEQ) as their label, and the other four arcs have s_mod(SET) as their label.

In the implementation, we again derive the oracle and testgraph classes for intset2 from those for intset1. For the oracle class, we add the state variable mod, which stores whether the current node is SEQ, or one of the original four nodes. The new oracle member function s_mod updates the value of mod. In addition, the new member function g_sum, implemented as a simple four-way case statement, returns the sum of the elements in the set.

The changes to the testgraph class are summarised below, where s is of type intset2 and s_o of type intset2_o.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>arc(s, s_o, a)</td>
<td>if a is TOSET or TOSEQ, calls s.s_mod and s_o.s_mod; otherwise calls intset1::arc(s, s_o,a).</td>
</tr>
<tr>
<td>nod(s, s_o)</td>
<td>if the current node is SEQ, calls chkseqmod and chkseq; otherwise calls chksetmod and intset1::nod(s, s_o).</td>
</tr>
<tr>
<td>chkseq(s, s_o)</td>
<td>checks the sum, number, and membership of the sequence of elements returned by sg_next as described above.</td>
</tr>
<tr>
<td>chkexcend(s, s_o), chkexcseqmod(s, s_o), and chkexcsetmod(s, s_o)</td>
<td>check that the exceptions END, SEQMOD, and SETMOD are signaled correctly.</td>
</tr>
</tbody>
</table>

Since we have changed the testgraph, we must also update the graph traversal. In this case, we use the same two paths as before, but we add the three calls

arc(s, s_o, TOSET); nod(s, s_o); arc(s, s_o, TOSEQ) after every node.

6 CONCLUSIONS

Testgraphs provide an abstraction of a class implementation, and are typically simple and intuitively appealing. We use testgraphs to drive input generation and to support test oracle development. Because full oracles are typically unaffordable, it is critical to design the testgraph and the oracle together.

The implementation of testgraphs for base classes is straightforward, and by varying the graph traversal algorithm and the graph parameter, we can generate many useful tests from a single testgraph. To test derived classes, we use the inheritance mechanism of C++ to build a test hierarchy that is isomorphic to the implementation hierarchy. For simple de-
rived classes, it suffices to add increments to node or arc labels of the base-class testgraph. For other classes, nodes and arcs can be added to the testgraph.

In the near future, we plan to simplify the implementation of testgraphs in C++. In particular, we will simplify the definition of nodes, arcs, and their labels, and automate graph traversal. In addition, while we currently use a customised method for signaling and handling exceptions, we plan to incorporate the C++ facilities for exceptions into the testgraph framework. We have tested a variety of base classes with testgraphs (Hoffman and Strooper, 1993), but more experience is needed with testing derived classes. For this purpose, we have started applying testgraphs to test some of the classes from the Booch C++ Components class library (Booch, 1991) and the CSet++ Collection Class Library (CCCL, 1993). Finally, to make testgraphs more generally applicable, we must address other object-oriented issues such as generic classes and overloading.

ACKNOWLEDGEMENTS
This research was supported by the Natural Sciences and Engineering Research Council of Canada under grant A8067.

REFERENCES
CCCL (1993): CSet++ for AIX/6000 — Collection Class Library Reference, IBM.


BIOGRAPHICAL NOTES
Dr Daniel Hoffman received the BA degree in mathematics from the State University of New York, Binghamton, in 1974, and the MS and PhD degrees in Computer Science in 1981 and 1984, from the University of North Carolina, Chapel Hill. From 1974 to 1979 he worked as a commercial programmer/analyst. He is currently an Associate Professor of Computer Science at the University of Victoria (British Columbia, Canada). His research area is Software Engineering, emphasising the industrial application of software specification and testing. He has published more than 20 technical papers in refereed journals and conferences, and has taught Software Engineering in academia and industry for the past eight years. He spent the 1992-93 year on sabbatical at Tandem Computers, Inc.

Dr Paul Strooper is a Lecturer in the Department of Computer Science at the University of Queensland. He received his PhD degree in Computer Science in 1990 from the University of Victoria. From 1990 to 1992 he worked as a Research Associate for the Institute for Robotics and Intelligent Systems at the University of Victoria. His research interests include software engineering and logic programming.

PARFETT, M. What is DIP? — a guide to Document Image Processing, NCC Blackwell, 128, and 118, pages respectively, $49.95 each.

These two titles form part of the British National Computing Centres EDI catalogue along with “The EDI Implementor’s Handbook”, “EDI and the Law” and “The EDIFACT Standards”.

Martin Parfett is a regular contributor to the series and has the rare talent of both understanding the subject matter and being able to relate that to others in a clear and simple descriptive form.

I was initially intrigued about the connection between these two technologies, but the author manages to tie them together convincingly. Indeed, I didn’t know what DIP was so I guess I needed the book!

Both technologies deal with the processing of business exchanges and the ability to capture information in a structured form. DIP and EDI work together to create the “paperless trading” world. DIP is that technology that is often referred to as Document Image Processing, Electronic Document Management Systems or Digital Optical Storage Systems. That is the optical or image capture of business document and the processes of these as structured data.

In “What is EDI?”, the author provides a concise explanation of the background and concepts, technology and issues involved in using EDI. This is complemented by frequent case studies and practical user experience. It was interesting to read the failed case studies as well as the glossy ones usually presented. Mr. Parfett presents a “warts and all” picture of EDI.

Despite the fact that this text is already outdated in some of its market descriptions and that it focuses strongly on the UK and EEC situation, it still provides many valuable insights and examples which are relevant to all of us.

“It is predicted that EDI will revolutionise the way companies do business. This is a bold statement to make if one considered EDI as just the development of another computer-based system. Certainly, such sweeping statements have not been made for payroll, order processing or computer-based systems. But one crucial difference between these systems is that EDI is a co-operative system, shared by trading partners and it is this major difference that underlines many of the reasons for EDI’s dramatic impact on business. — (What us EDI?, p17).

These texts do not qualify as technical or academic. They provide a refreshing light yet comprehensive summary of the basics of these exciting new applications of technology.

Having some knowledge of one subject matter (EDI) and none of the other (DIP), I can say that I found the EDI text covered all the basics that managers, consultants or users may need to know. I hope that as a DIP novice I have gained a similar introduction to that topic.

I would certainly commend these titles as good background library texts which should be readable by technical and non-technical management alike.

Tim McGrath
Fremantle, WA


“This book is intended to introduce the reader to the topics of Quality Management and to explain how it is applied to software and IT projects” — from the Preface.

This slim volume (121 pages, plus a series of Appendices) attempts to meet this goal, and proceeds with the best of intentions. Following a general introduction, it addresses the issue of defining software product quality. It introduces a number of high-level user characteristics of software in a sound, well-structured approach, but does not adequately address the issues of measurement of these characteristics, or of how the product may be engineered to achieve them. The author’s development probably preceded the release of the International Standard for software quality evaluation, ISO9126, so it is not surprising that the discussion does not adopt the standard framework; however, more reference to other work on quality engineering would have been helpful.

It then presents a clear, rather audit-oriented discussion of the implementation of a quality management system (from the UK perspective — all the references are to BS5750, their version of ISO9110). There is encouraging emphasis on the need for software inspections, but once again, true depth of software engineering theory is lacking. It is surprising in the current context, to find a book discussing software quality and not specifically addressing issues of process capability and management. The discussion of quality improvement, in the last chapter, continues this approach, and is disappointing in its lack of depth and insight.

The book is written from the perspective of an immature software organisation. As such, it may be of use for managers and practitioners in small organisations beginning to feel their way towards quality and keen to improve their development capability. Experienced professionals, however, will find it somewhat lacking depth and insight; the approach overall can best be described as superficial.

Terry Rout
Griffith University


This book was not exactly what I thought initially, judging from the title alone. No, it’s not a book on parallel algorithms or the theoretical and practical sides of parallel computing. Wrong title or wrong book? As the preface states, the book is a modification of Golub and Ortega (1991). So don’t hold your breath when you’re looking for a new book on parallel computing. In about forty pages, parallel architectures and programming techniques are covered. Many of the advanced parallel methods are contained in better and more graphic books on parallel algorithms such as Thompson Leighton’s (1992).

Mind you, the book is an excellent textbook as an “introduction to numerical methods”. It is written in a clear way. It contains an abundance of exercises and the right dose of parallelism for an engineering, science or computer science undergraduate studying scientific computing with access to a parallel computer. Throughout the book, there are, more or less hidden extensions, in the form of added subsections, on parallel variants of the sequential algorithms presented earlier.

The first three chapters are introductory, the “world of scientific computing”, basic reference material on “linear algebra” for a quick review or later, and “parallelism and vector computing” in a nut shell. All in 90 pages.

The body of basic numerical approaches and methods follow in three super chapters. “Polynomial approximations” covers first course material such as Taylor series, least squares, roots — in about 50 pages. “Continuous problems solved discreetly” (76 pages) motivates and elaborates discretisation of differential equations by linear systems of equations. “Direct solutions of linear equations” (another 60 pages) studies Gaussian elimination with extensive error handling and a variety of factorisations.

The final three chapters each more or less 45 pages, cover more advanced material including some parallel and vector machine implementations, partly hidden in subsections. “Parallel direct methods” presents methods of the previous chapter revised for parallel and vector machines. “Iterative methods” focuses on relaxation-type (e.g. Jacobi, Gauss-Seidel and SOR) and multigrid methods. The final chapter covers “Conjugate gradient-type methods” for both symmetric and nonsymmetric linear systems of equations.

In summary, an excellent text for lecturers and students of numerical methods. Even if parallelism is hidden in the text more than in the title, it is just right for such a course.

References:

Heinz W. Schmidt
CSIRO and ANU


This book, currently in its sixth edition, is suitable for either the novice or experienced user of DOS. Divided into three parts, the first section, “Getting to know MS-DOS”, has been written explicitly for the raw beginner, and as such introduces the concept of an operating system followed by the basics for using MS-DOS. In section two, “Learning to use MS-DOS”, the book
explores MS-DOS 6.0 in depth, highlighting the new features to be found in this latest release. The final part of the book contains three appendixes. Appendix A covers the installation of DOS (versions 3 through 6), appendix B is a glossary of frequently used DOS terms, whilst appendix C is an alphabetical listing of all available DOS commands. This appendix is a valuable reference tool, concisely bringing together the information needed to use a command: what it does, the versions of DOS that it is applicable to, a description of its appropriate parameters, examples of how to use the command, plus a reference to the body of the text where this command was first explained.

"Running MS-DOS" describes in detail all of the new features to be found in version 6. For example, just some of the new commands and programs are as follows. "DELTREE" moves a file from one location to another. "DELETESPACE" compresses all the files on a hard disk. Another new program is "DEFrag" which packs all the files into a single compact region. The text of the book is well-presented, with screen shots and diagrams added where necessary. The author has incorporated highlighted notes where further clarification is needed.

"Running MS-DOS" allows you to maximise your use of the DOS 6 operating system. It shows you how to do tasks you didn't think were possible or didn't know how to do. It is written in a tutorial style with plenty of examples, which encourages a hands-on approach to learning. It is a valuable user's guide to DOS and deserves a place on your bookshelf, but you will need to add to it if you wish to do advanced DOS programming.

A. Stevenson
Blackburn, Victoria


This book is somewhat novel in its approach to the topic in that it attempts to cover distributed computing in its broadest sense. The book begins by defining some terms and establishing the framework for the rest of the book. Once the framework is in place, Umar deals with networks covering LANS, MANs, to WANs. The OSI model is examined as are standards such as ISDN, TCP/IP and SNMP. One glaring omission is the lack of discussion of ATM.

After dealing with networks, Umar turns his attention to application support services required in distributed computing. Topics covered here include various examples of client-server systems, remote procedure call and sockets. Distributed database management is discussed as are distributed operating systems with some examples such as Amoeba and Athena.

The final three chapters of the book deal with distributed applications and downsizing. Umar introduces a methodology here to assist the reader in the development of distributed applications.

The book ends with five tutorials of various topics: Communication protocols and the OSI stack; TCP/IP; SNA; MAP and TOP; and SQL. Although these tutorials are short, they provide a level of understanding that is adequate, but not deep.

Given the ambitious nature of the book, it must be said that Umar has done a good job. He has covered all aspects of distributed computing from the network level down to the application level. Some topics are not covered in as much depth as the reader might like (or need), but the reference list provides valuable pointers to detailed and specific information.

The index is somewhat disappointing, limiting the books usefulness as a reference text. However, the reference lists provided at the end of each chapter are comprehensive and provide the reader with valuable sources of additional and more detailed information. The exercises and case studies at the end of each chapter and tutorial give the reader a chance to practice what they have learned. The case studies are fairly open ended allowing the reader to explore the possibilities to as much depth as desired.

Generally, the book is good for professionals wanting to update their knowledge of this important area. It is useful for academic courses provided that the course is structured in a similar manner to the text. A course on a topic such as networks would do better to use a book which deals solely with that topic — a by-product of the fact that Umar has probably attempted to cover far too much in the space of a single book.

M. Oudshoorn
University of Adelaide

Blackwell, 344pp., $39.95 (paperback).

Accounting applications form the backbone of many organisations' data processing operations. Most commercial pioneers of computing developed their accounting applications first, while the VISICALC spreadsheet was seen as being the 'killer application' that ensured the corporate success of personal computers. Whether it be a venerable general ledger system (circa 1975, and typically written in 'spaghetti' COBOL) in large companies or the corporate spreadsheet in a small business, computing professionals in the commercial sphere are highly likely to encounter accounting applications.

It is in this context, then, that this book would be a highly useful addition to the library of most applications developers, particularly those who have recently graduated from courses where the exposure to accounting concepts may have been minimal. The first section of the book provides an organisational context for accounting, and a conceptual overview. The mechanics and conventions of bookkeeping are covered in the second section. Section three deals with control and audit requirements, while Section four deals with cost accounting. Although the book may sound like a standard accounting text so far, it is differentiated by the fact that throughout these four sections reference is made to the implementation aspects of these functions on computerised accounting systems. There is a good section, for example, on the particular challenges of auditing computerised systems as opposed to manual ones. The final section, section five, deals specifically with computers and accounts, including system configurations and spreadsheet functions. It is this last aspect, i.e., spreadsheets, that I believe the author could have expanded on a little, given their prevalence today.

Throughout the book there are exercises for the reader, together with answers in the appendix. This makes the book a good choice for self-education, as well as a student text. The book has obviously been successful in the past, as it is now in its third edition.

Mike Riddiford
RMIT

Blackwell, UK, 130pp., $69.95 (paperback).

I found myself agreeing with the author when reading the Preface, this made me happy to continue reading the rest the book, and I was not disappointed. O'Brien stresses that management in a business sense is always about people and software production management should be regarded in the same light.

There are four parts to the book:
1. Environment. This includes the nature, costs and benefits of information, the demand and supply of software products and some of the problems businesses are facing because of inadequate and costly software products.
2. Technology. This section covers the nature, production, maintenance, quality and metrics of software and the technological tools that have been developed with a view to increase production and hopefully reduce costs.
3. Psychology. An amusing description of "real" programmers and their culture is included, with reports from several surveys. O'Brien does warn the reader in the introduction, that they may feel depressed after reading this part.
4. Renaissance. The reader is to be brought out of the dark ages, not by some new and mind blowing solution, by looking outside the software production area at successful solutions to other seemingly unrelated problems. For example, if a software product is thought of as a service, concepts relevant to service industries can help to rethink the way customers are regarded. The type and content of training is considered, and the results of a survey by the British Institute of Management are cited as being of relevance to the training of software managers and other workers in software production, though their original concern was of general business managers.

A checklist of steps that might be taken when developing a strategy for software production management is included, followed by a presentation of the author's conceptual framework which establishes relationships between the "needs" of the elements of an organisation.

There is a good "wide-angled" bibliography, which includes references made in the text.

Despite being rather expensive for such a slim volume, I recommend that this is essential reading for those involved in software production as well as those teaching and/or studying related subjects in tertiary institutions.

Liz Haywood
Lecturer, Victoria University of Technology
AUSTRALIA, THE CLEVER COUNTRY, HELPS HP

Hewlett-Packard Australia and Queensland software company, CITR, recently announced the formation of a strategic R&D relationship. CITR is set to undertake software architecture design and development for future releases of the HP OpenView network and system management platform. The CITR R&D deal is worth more than $5 million in the first phase of the project.

CITR was chosen by Hewlett-Packard to conduct this strategically important work due to its world-class expertise in the operations and applications of HP OpenView.

CITR is a high-technology software development company owned by the University of Queensland’s holding company, UQ Holdings. The company specialsises in leading-edge technologies spanning the worlds of computers and communications and as such is well positioned for the emerging Information Superhighway scenario. The company had 1993 revenues of more than $5 million.

The announcement has met with enthusiastic response from both Federal and State leaders.

Federal Industry Minister, Senator Peter Cook, welcomed the decision by Hewlett-Packard to work with CITR to develop key software in Australia for its global needs.

Queensland Premier, Wayne Goss said the HP-CITR relationship presented a major step in Queensland’s goal for the Sunshine State to become the smart industry centre of the southern hemisphere.

The CITR deal is an integral part of HP Australia’s Partnership for Development Program and is further evidence of the company’s commitment to promote Australian intellectual property within its global product charters. As a result CITR will work closely with HP’s Network and Systems Management Division (NSMD) based in Fort Collins, Colorado. NSMD’s mission is to develop a broad range of network and systems management products in an open systems environment.

The agreement is open-ended and the HP-CITR relationship has the potential for significant growth.

“HP’s future in the systems business is reliant on the success of OpenView, particularly in the telecommunications market, and the CITR agreement is evidence of our belief that Australia can help us meet our objectives”, said HP Australia Managing Director, Bill Hilliard.

The Director of CITR, Rob Cook, also a former editor of the Australian Computer Journal, met with HP’s Chief Executive Officer, Lew Platt, during his April visit to Australia to discuss the relationship.

“CITR has become an important supplier of management and distributed software solutions and technology in Australia”, Mr Cook said.

“This relationship gives us an export focus and will help us to grow rapidly, providing products and technology to the HP OpenView and other management software marketplaces worldwide”. We are delighted to be working with HP and look forward to deepening and broadening this groundbreaking relationship”.

CITR was formed in 1986 as an independent, commercial organisation to bring new technology solutions to the marketplace. CITR has specialised in applying open distributed system and OSI technologies to the development of platforms and end-user applications, with an emphasis on telecommunications network management. The company has been involved in the development of several international standards for remote database access, distributed transactions processing and open distributed processing.

HP OpenView offers users integrated network and system management for multivendor distributed-computing environments. Solutions consist of a broad portfolio of management applications from HP and OpenView Solution Partners, and a complete set of services that help customers improve service and reduce network operation costs. OpenView manages Novell NetWare PCs, and IBM, Sun Microsystems, AT&T, Bull and HP computer systems.

OpenView is installed in more than 35,000 networks around the world. There are more than 200 complementary OpenView-based network and system-management solutions available on a variety of platforms, including HP 9000 systems, Sun Microsystems Solaris and SunOS workstations and Microsoft Windows PCs. OpenView is licensed at AT&T, Data General, Hitachi, Groupe Bull, Alcatel and Siemens.

STATE OF THE ART INFORMATION SERVICE TO LINK NATION’S LIBRARIES

The National Library of Australia will introduce a new National Document and Information Service which will link users of Australia’s libraries to New Zealand and the rest of the world, the Minister for Communications and the Arts, the Hon. Michael Lee, announced recently.

The National Document and Information Service (NDIS) will be a leading edge system for on-line access to material in libraries throughout Australia, with gateways to overseas services. It will be developed in co-operation with the National Library of New Zealand on a partnership basis and marketed to users in both countries.

“This project presents an exciting opportunity for the National Library and the Australian library community to exploit recent advances in technology and take advantage of the development of the Internet and related broadband services. It also provides an opportunity for a dynamic partnership between Australia and New Zealand government agencies with similar objectives Mr Lee said.

The NDIS project will replace older systems such as the Australian Bibliographic Network (ABN) and other bibliographic and database services. This venture will provide a user-friendly system which will make it possible for a greater number of Australians to gain access to a wider range of information services.

Users of the new system will include libraries, the corporate sector, institutions such as hospitals and schools, and individuals who need information not available locally. Anyone with online access, be it at their workplace, place of study or home, will be able to search NDIS to identify and locate books, journal articles and other documents. The system will allow quick and easy access to information and facilitate fast delivery of material either in printed or electronic form.

The national libraries of Australia and New Zealand will be entering into negotiations with CSC Australia (formerly Computer Sciences of Australia) as the preferred vendor for the NDIS project. CSC is a major systems integration and software development company, and is a member of the Australian Government’s Systems Integration Panel.

The new service is scheduled for delivery by 1996.