A Framework for Exploiting Emergent Behaviour to capture 'Best Practice' within a Programming Domain

"This thesis is submitted in partial fulfilment
of the requirements of the award of Doctor of Philosophy
of the
Oxford Brookes University"

Sarah Jane Mercer
September 2004.

Collaborating establishment:
Her Majesty's Government Communications Centre

Abstract

Inspection is a formalised process for reviewing an artefact in software engineering. It is proven to significantly reduce defects, to ensure that what is delivered is what is required, and that the finished product is effective and robust.

Peer code review is a less formal inspection of code, normally classified as inadequate or substandard Inspection. Although it has an increased risk of not locating defects, it has been shown to improve the knowledge and programming skills of its participants.

This thesis examines the process of peer code review, comparing it to Inspection, and attempts to describe how an informal code review can improve the knowledge and skills of its participants by deploying an agent oriented approach.

During a review the participants discuss defects, recommendations and solutions, or more generally their own experience. It is this instant adaptability to new

information that gives the review process the ability to improve knowledge. This observed behaviour can be described as the emergent behaviour of the group of programmers during the review.

The wider distribution of knowledge is currently only performed by programmers attending other reviews. To maximise the benefits of peer code review, a mechanism is needed by which the findings from one team can be captured and propagated to other reviews / teams throughout an establishment.

A prototype multi-agent system is developed with the aim of capturing the emergent properties of a team of programmers. As the interactions between the team members is unstructured and the information traded is dynamic, a distributed adaptive system is required to provide communication channels for the team and to provide a foundation for the knowledge shared. Software agents are capable of adaptivity and learning. Multi-agent systems are particularly effective at being deployed within distributed architectures and are believed to be able to capture emergent behaviour.

The prototype system illustrates that the learning mechanism within the software agents provides a solid foundation upon which the ability to detect defects can be learnt. It also demonstrates that the multi-agent approach is apposite to provide the free flow communication of ideas between programmers, not only to achieve the sharing of defects and solutions but also at a high enough level to capture social information. It is assumed that this social information is a measure of one element of the review process's emergent behaviour.

The system is capable of monitoring the team-perceived abilities of programmers, those who are influential on the programming style of others, and the issues upon

which programmers agree or disagree. If the disagreements are classified as unimportant or stylistic issues, can it not therefore be assumed that all agreements are concepts of "Best Practice"?

The conclusion is reached that code review is not a substandard Inspection but is in fact complementary to the Inspection model, as the latter improves the process of locating and identifying bugs while the former improves the knowledge and skill of the programmers, and therefore the chance of bugs not being encoded to start with.

The prototype system demonstrates that it is possible to capture best practice from a review team and that agents are well suited to the task. The performance criteria of such a system have also been captured.

The prototype system has also shown that a reliable level of learning can be attained for a real world task. The innovative way of concurrently deploying multiple agents which use different approaches to achieve the same goal shows remarkable robustness when learning from small example sets.

The novel way in which autonomy is promoted within the agents' design but constrained within the agent community allows the system to provide a sufficiently flexible communications structure to capture emergent social behaviour, whilst ensuring that the agents remain committed to their own goals.

Table of Contents

Table of Figures

Table of Tables

Acknowledgements

Firstly a big thanks to my supervisors, Sue, Howard and John, without whom I would have gone completely and utterly mad! Secondly a huge thanks to my family for their support, especially Paul for all the cups of tea. And lastly to all my friends and colleagues, in particular Mark F, Pete T, Pete D and Richie, who all contributed to some of the crazy discussions on agents, machine learning, and the magic of code review. And not forgetting the testers who desperately tried to break it – cheers guys!

Arthur C. Clarke's Third Law of Prophecy:

"Any sufficiently advanced technology is indistinguishable from magic".

Chapter 1

Introduction

Enterprises in all developed sectors of the economy are increasingly dependent on software-based systems. From the automotive industry to consumer electronics, not only is software being incorporated into products but also into the support management, production and service functions of these diverse organisations. Given this, the need to produce software efficiently, effectively and with consistent high quality is becoming increasingly important for all industries [1, ESSI]. Products which rely heavily on software are no longer uncommon or unusual in everyday life. Even in high reliability safety critical environments such as the vehicle, software is used to control engine function, fuel supply, anti-lock braking system, diagnostics etc. Some new models use "fly-by-wire" technology, where everything is translated from physical action to computer instruction for processing and back to physical

action. This greatly increases the requirement for software to be correct, reliable and robust.

Much work has been carried out over the last thirty or so years to increase quality within software development [2, IEEE][3, BCS][4, ACM], to move it from being perceived as an art form and towards a structured engineering discipline. But software still suffers from bad press; there have been a number of recent, highly publicised, government funded software projects, which have either failed, been substandard, or significantly over budget (MOD projects [8, Arnott], Child Support Agency [9, BBC], Payroll for Prison Service [10, Arnott], Air-traffic control [11, Ranger], [12, Computing], [13, Arnott], [14, Arnott]).

Whatever the project, the problems/bugs/defects/anomalies can be categorised as follows: human errors (designer's mistake), faults (encoding of an error into a software document / product), and failures (deviation of the software system from specified or expected behaviour).

A significant amount of research is currently being undertaken in these areas [15, QAI], [16, Software QA/Test Resource Center], [17, SEI], [18, SQI], [19, SATC], [20, ESI]. New methodologies are introduced as revolutionary methods that are supposed to solve the software crisis. These mostly focus on fault avoidance, and most claim "theirs is best". Empirical evidence for formal methods and Object Oriented Design does not support the claims made. Even the claims made about structured programming are not conclusively supported by the evidence. It is important to note that this same empirical evidence does recognise that inspection techniques are cost-effective. Although ill-defined [ESSI, 2001] some research even

suggests that inspection techniques are more cost-effective than structural testing [21, Laitenberger], for example, it has been observed that reading / inspection can identify 1 defect an hour, compared with 0.3 defects an hour for black and white box testing [ESSI, 2001]. It is also important to note that the same evidence shows the defects identified by inspection are complementary to those found by testing. Testing is critical to the development of robust and reliable software, but is outside the scope of this work.

Fagan described the Inspection (with an 'I') approach in the software domain, more than twenty years ago [5, Fagan]. More recently many others such as Gilb [6, Gilb & Graham][7, NASA] have fine-tuned the Inspection method to make it an even more cost-effective instrument for tackling quality deficiencies and defect costs. It has been claimed that Inspection can lead to the detection and correction of between 50 and 90 percent of defects [22, Laitenberger]. There are many points within the software lifecycle where a defect detection phase can be used, for example, in ensuring that customer requirements are consistent with the problem description, or that the detailed design conforms to the architectural design. The goal of a defect detection phase is to identify defects through scrutiny of a software artefact; however the way in which this should be organised is still debated. Study of inspection techniques and small group psychology by Laitenberger of the Fraunhofer Institute for Experimental Software Engineering has led him to recommend that defect detection activity can be organised as both individual and group activity with a strong emphasis on the former.

Peer code review is the preferred format of inspection at HMGCC. Less formal than the process described by Gilb, it involves a review of the source code produced, not

as a validation exercise (conformance to design), but as a verification that the code produced is correct. Each reviewer inspects the code and identifies defects individually. The reviewers then gather to discuss findings and agree recommendations for change.

There is strong empirical evidence that this style of review not only identifies bugs (faults) and promotes consistency, but also stimulates learning [22, Laitenberger]. New and innovative ideas are proposed by the authors, the reviewers, based on experience, spot faults and provide remedial solutions. If any new guideline or elegant solution is identified, feedback is given to the whole team who then quickly adapt to the new knowledge. These positive outcomes are not controlled or managed. The coding guidelines are not written down or published - they are simply remembered as "tribal knowledge". Any attempt to capture the process and/or its output has been cumbersome, with the results vague, constricting and very soon out-of-date. This unexpected behaviour could be described as the emergent behaviour of the code review process within the team environment. It is this valuable new knowledge that we want to capture and use.

Within the field of Artificial Intelligence a relatively new area of research has shown that emergent behaviour can be captured; agent technology. It has been proposed that multi-agent systems can exhibit emergent behaviour [32, Luger & Stubblefield]; "the specification of the behaviour of the agent alone does not explain the functionality that is displayed when the agent is operating" [33, Maes]. Some researchers categorise this as a "pitfall of agent oriented development" [37, Jennings and Wooldridge]. However, both agree that this behavioural phenomena occurs as a consequence of the interaction between the components in a multi-agent system.

Pattie Maes goes further to say that "the functionality of an agent is viewed as an emergent property of the intensive interaction of the system with its dynamic environment". Although it is said that agent systems are capable of demonstrating emergent behaviour, it is the emergent behaviour generated from the interactions between the human software developers, that the agents are attempting to capture.

"Autonomous agents" have been proved useful in applications where adaptivity is required. The user is engaged in a cooperative process in which human and computer agents initiate communication, monitor events and perform tasks. Over time, the agent or personal assistant becomes increasingly more effective as it learns a user's habits and preferences [38, Maes]. The ability to learn represents one of the most important criteria for intelligent operation [39, Brenner, Zarnekow & Wittig]; however, some consider learning in a multi-agent system to be a risk [42, Malone, Lai & Grant], as it is feasible for a system to infer incorrect rules (or fail to infer correct rules).

If considering a system that incorporates learning and emergent behaviour, the risks associated with such ideas must also be considered. To mitigate, or at least minimise, the risk of erroneous learning, the agent as an advisor is equally placed in the advisee role; so the programmer can give feedback to reinforce or refine the agent's knowledge. Above this, the agent is not permitted to modify the user's code; it is only advising the user of the changes to be made. Limiting the risks of emergent behaviour is more abstract, as we are unable to specify it in the design; we are equally unable to design the risks out. However, the way in which the agents communicate, although unstructured, can be controlled by limiting the performatives

they use, the agents with whom they can interact, and by giving each agent a clearly defined role and set of objectives.

This thesis introduces a system that attempts to capture the emergent behaviour of the code review process. A hypothesis is presented which demonstrates that by modelling the team as an adaptive multi-agent system, the system can mimic the human emergent behaviour using relatively simple reactive agents. It is proposed that this system, although only a prototype, has important theoretical implications; that an effective and reliable standard of learning can be achieved in the real world, by incorporating congruency into a learning system. It is this congruency, coupled with the innovative way in which the data is presented to the learning system, that gives this learning system credibility.

This thesis starts by analysing the need for a system to advise on good programming practice, evaluates some static-analysis tools that are currently available, and discusses why the proposed system is better (Chapter 2). The best way to design and implement such a system is presented in Chapter 3 together with a review of the capabilities of agent technology and some example systems which are able to assist the user without specific direction or specification. The agent architecture is then described and justified (Chapter 4). The basic learning mechanism is introduced and evaluated (Chapter 5). Later work demonstrates how learning is improved by the inclusion of relational data (Chapter 6). The results of testing are then presented, evaluated and discussed (Chapter 7). A discussion of the system, its significance and limitations is presented (Chapter 8) along with possible directions for future work.

1.1 Contributions

It is the aim of this thesis to determine, a) if it is feasible to capture and automate best practice, b) the way to provide this (agents) and c) the method by which such a system could:

i. Act as a consultant on best practices.

ii. Learn the fundamentals of good programming practices.

Chapter 2

Good programming practice

This chapter discusses how bugs originate in an attempt to understand where a system to prevent bugs would be best situated (section 2.1). It also compares the relative strengths and weaknesses of formal code Inspection and peer code review in an attempt to identify beneficial aspects of these processes (section 2.2). From this are formed a number of requirements that a system must satisfy if it is to assist the user by advising on good programming practice.

Existing static analysis products, both from industry and academia, are then reviewed and the findings presented (2.3). The system requirements and the review of current tools allow us to present a detailed analysis of requirements (2.4).

2.1 Where do bugs come from?

The implementation stage of the software lifecycle, where design is translated into a machine-readable form, is often trivialised in design methodologies and referred to as code-generation. By assuming that the design is performed in a detailed manner the process is perceived to be mechanistic [27, Pressman]. However, in the real world this is not always the case, and the distinction between design and implementation is often blurred. A design may not be refined to the level where it can be directly translated to a computer language. It may be decomposed to units which are the equivalent of a "head full", with the understanding that the programmer will refine the design further whilst implementing it. Decisions are made about issues that were overlooked or deferred by the design process, such as race conditions or other target specific issues that are not completely dealt with in the data (SSADM) or event (UML) centric methodologies.

So where do bugs come from? Some believe it is the pressure applied during development in the rush to get products to market without adequately testing them [29, Thibodeau] (1). Some believe it is the lack of personal best practices: while software developers usually get extensive training on tools and methods, they generally get little or no guidance on their personal practices, on actually how to manage their coding [28, Humphrey] (2). Others simply believe that the task of programming is difficult, in a long program, perhaps consisting of thousands of lines of code, it is possible for the writer to misjudge how the various elements of the software will work together [30, Newscientist] (3). Bugs can also be introduced simply as an error.

If the problem is divided up and designed into units of a "head-full" it is more reasonable to assume that a number of errors will be generated from the interfaces between these units. Inadequately specified interfaces can lead to unusable or bug ridden units. Head-fulls can reduce the complexity (3) by allowing the programmer to only be concerned with the elements within his unit. However a number of risks can be introduced. For instance, the size of a unit differs between programmers with varying levels of skill and experience.

There have been a few attempts to cognitively understand the problem and model the programming process, with the aim of answering this question. In summary, the task of software development is made difficult because of the interplay between the formal world of the computer and its programming language and the informal world where the problem is to be solved [31, Jackson]. The task of programming is difficult because it is problem solving in multiple problem spaces: rule, instance and representation. Programmers generate or refine programs in the rule space, test the programs in the instance space, and search in the representation space for solutions that lessen the cognitive difficulty of program generation and / or program testing in the previous two spaces. This representational change has been identified as one reason that programming is (sometimes) difficult [34, Kin & Lerch][35, Walenstein]. So ironically some bugs in software stem from the interactions of a complex system within a complex environment - the software development environment's emergent behaviour.

2.2 Software Inspection 'v' Peer Code Review

Some coding errors can be detected and rectified by the use of existing software engineering methodologies, such as guidelines for good design (high-cohesion and low-coupling) and implementation (data-abstraction and data-hiding). Also, the quantitative assessment of conformance to these guidelines can be measured by the use of Software Metrics that provide an indirect measure of quality, such as Halstead's software science and McCabe's complexity metric [43, Pressman]. However, such formal and mathematical approaches are limited in that they are only as accurate as the information (model) supplied, and are therefore equally susceptible to falsities and human error [23, Butler & Johnson].

As discussed in the introduction, software inspections can significantly and cost-effectively improve the quality within software systems, by detecting defects early in the lifecycle and therefore reducing the associated cost impact. As also mentioned, defects identified during code review are different from those found by testing; it is assumed that code review is supplemental to a testing strategy when developing high quality, reliable systems.

The less formal process of peer code review involves an examination of source code, with the focus being on how the code is constructed and not on what it is intended to do, normally facilitated by checking the code against a common set of standards. A number of reviewers are involved; each one inspects the code and identifies defects independently, then the group meet with the author to discuss findings and agree recommendations for change.

The defects identified normally fall into one of four categories: 1) choice of representation; 2) adherence to standards and programming language conventions; 3) choice of strategies for error handling and debugging; and 4) awareness of target-platform issues.

Previous studies have shown that this style of review not only identifies bugs and improves consistency, but that it is also capable of improving the programming skills of its participants. Any new and innovative approach is shared, as too is the experience by which reviewers spot faults and provide remedial solutions. The review meeting provides an ideal opportunity to not only share this knowledge but also to adapt by identifying new guidelines and solutions.

It is this swift adaptivity that is not supported by Gilb's framework for Inspections, as the standard to which code should be written is also a software product, and as such is subject to scrutiny under Inspection. The speedy absorption of new information is stifled in the formal environment of software Inspections. Another difference between the approaches is that Gilb's framework does not attempt to capture solutions; it does not attempt to identify the best suited solution for a found defect.

As presented in Figure 1, the Inspection method is capable of improving the process of inspection and the standards to which code is inspected. The programming ability of the author and the reviewers is not a concern of the process, only the detection of defects. The programming experience of the author and reviewers is essential within the code review process. Code review is able to improve the process of

programming. It is able to swiftly adopt and share (between the review team), new information, including solutions.

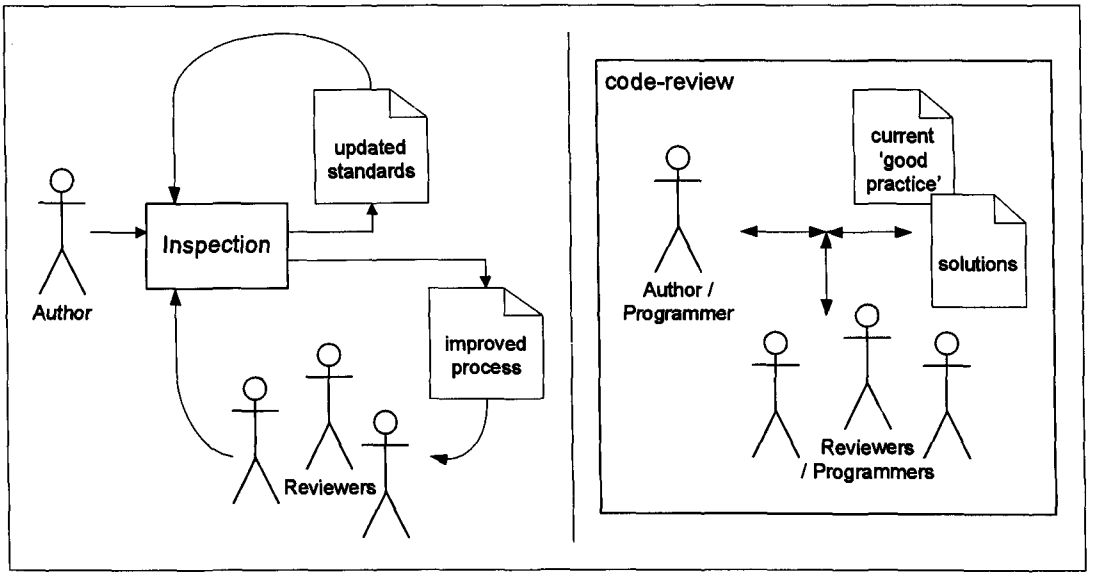


Figure 1: Inspection and code review processes

The positive outcomes of the peer code review process are not controlled or managed, the coding guidelines are not written down or published, they are held as "tribal knowledge". Any attempt to capture the process and or its output has been cumbersome, with the results vague, constricting and very soon out-of-date.

It is interesting to note at this stage that the main strength behind the code-review process is the team of engineers. It has been shown at HMGCC that when using a team of 3-4 people (1 author and 2 or 3 reviewers, one of which is the moderator) the number of anomalies pinpointed is significantly greater than a review with just 1 other engineer. There has been a great deal of research on the optimum number of reviewers, but no consensus has been reached [6, Gilb & Graham][26, Porter et al][22, Laitenberger][5, Fagan]. Laitenberger recommends a team of 3 or 4: one author, one or two inspectors (reviewers) and one moderator. It is worth noting that

there is empirical evidence to show that the effectiveness of each individual reviewer can vary greatly. Some engineers are better at identifying complex bugs such as race conditions, others are more focused on the error handling strategy, and some others look to ensure the interface to the operating system is adhered to.

Another strength of the process at HMGCC is the use of two-phase reviews, where easily identifiable defects are found and corrected before the 2nd phase of the review is conducted. (This was adopted from the world of publishing, where a manuscript is first checked to fix spelling, punctuation and grammar, thereby allowing the second phase to concentrate on the content.) The first phase of the code review can be conducted by a single reviewer (possibly using a static analysis tool, if the author has not already done so). The defects it is concerned with are those within limited scope, e.g. variable naming, magic numbers, un-initialised variables, etc.

As with software Inspections, peer code reviews are resource intensive and are therefore commonly held at the end of the stage, when the code is considered finished by the author. This means that defects introduced by the author are not identified until the end of the stage, and the opportunity to learn from these mistakes is deferred. One way of providing feedback to the developer during coding and reducing the number of defects in the code before inspection is the use of static analysis tools. Such tools are encouraged, as they are very effective at finding certain types of defects, allowing the reviewer to concentrate on the other more difficult aspects of the inspection.

Coding is part of the creative process within the development of software systems. Experienced programmers prove invaluable during reviews in finding problems that

did not arise at design review and are not identified by normal means (such as the compiler / static analysis tools). This experience leads to the concept of 'Best Practice'. Using the peer code review process as a forum, this best practice is shared and refined.

2.3 Static Analysis & Automatic Assessment

There are a number of commercial automated software quality systems on the market, most of which concentrate on program correctness using a number of testing methods. A few applications use static analysis to inspect the code, a subset of which are described here: NuMega's CodeReviewTM, Gimpel Software's PC-Lint, and Parasoft's Code Wizard.

2.3.1 Static analysis tools

PC-Lint (a member of the LC-Lint family)

[55, PC-Lint] [56, LCLint]

This product has been used for a number of years in HMGCC, although not by everyone. The LINT family of software has a long-standing history, founded in ANSI C. PC-Lint covers the K&R, ANSI and ANSI/ISO standards which equate to approximately 880 rules. It has a powerful (flexible and complicated) script interface, where rules can be overridden for certain modules / function / variables etc. It is capable of analysing a whole project and / or just a single source code module, and reports bugs appropriately, i.e. when analysing a whole project, it can identify where memory has not been freed, where it is passed in and out of functions and across module boundaries. With time and effort an experienced programmer can

maximise the potential of PC-Lint as it is capable of a very thorough review, however only an experienced programmer would be pragmatic enough to avoid acting upon some recommendations that introduce complexity and intricacy to a project.

In a recent study [60, Hills] PC-Lint was used to quantify the effort required to review code to the MISRA standard [61, MISRA]. It was shown to cover 74% of all rules and 82% of the required rules, leaving only 12 rules which would need to be checked manually, as neither PC-Lint nor the compiler were able to do so.

Parasoft's Code Wizard

[58, Parasoft]

This product is very similar to PC-Lint in the way that it reviews the code, but the authors have tried to make it easier to configure, and in doing so it appears that they have reduced the functionality and consequently the usability. Code Wizard only analyses whole projects (which may be time consuming as it can take minutes to process). The number of rules supported is comparable to that of PC-Lint, but the amount of information given for each bug / issue found is not. The system occasionally gives an example of how to correct the bug, which can be very helpful.

NuMega's Code Review

[57, NuMega]

Although limited to Visual Basic, this system attempts to identify the more tricky bugs, such as target-specific ones. It also gives a good amount of feedback to the user as it exploits Microsoft's "knowledge bases" by redirecting the programmer to a

page in the MSDN (Microsoft Developer Network) that describes the bug / issue. This affords a decent amount of information as to why the recommendation was made and how to fix the bug. The system contains a knowledge base of over 400 rules, which is user-extensible, allowing users to enable, disable and edit alert messages.

The product caters for teams by allowing a centrally stored set of rules to be shared by all programmers, using a client-server architecture, to ensure that VB code produced is written to the same standard. Code Review reports on problems using a severity level for each. A set of metrics is gathered about the whole project to identify levels of complexity and future risks. It is fully integrated in the VB IDE, and allows the user to select the modules to be reviewed and select the types and severity of problems to find.

Summary

A summary of the capabilities of the Static Analysis Tools is given:

Product	Team aware	Effective review	Cope with changing requirements	User friendly description of bugs
PC-Lint	No.	8/10 - For coverage and usability.	5/10 - Can turn off rules in script for code artefacts.	7/10 - Explanations and reasoning behind guidelines are given.
Code Wizard	No.	6/10 - For coverage. Usability is not as good as PC-Lint.	6/10 - Sub-sets of guidelines can be disabled. Standards can be changed.	6/10 - Brief description is given, but sometimes an example of a solution is provided.
Code Review	Yes, standards can be controlled centrally.	7/10 - For coverage and usability.	4/10 - Individual guidelines can be enabled / disabled, locally and / or centrally.	8/10 - Uses MSDN's knowledge base for information about bugs.

Table 1: summary of review of static analysis tools functionality

In summary, static analysis tools are quite sophisticated, detecting a large array of bugs. PC-Lint is the preferred tool as its flexibility makes it extremely powerful and

usable, over and above other less flexible systems where good recommendations are hidden in repeated inconsequential noise that cannot be suppressed.

It appears from these systems and their relative strengths and weaknesses that being off-line but easily accessible would be preferable. Part of Code Review's charm is that it is so well integrated into the development environment that it is readily to hand. The amount of information supplied when a defect has been found is also important. A mixture of all three systems would be ideal. Good explanations / justifications of the guideline are necessary, the use of other resources such as the MSDN is highly desirable, as is the ability to suggest solutions or give good examples of fixes.

2.3.2 Automated Assessment

There have been a few systems developed to automatically assess and report on source code, mainly driven from the academic scenario of coursework for programming courses, where the mark and feedback needs to be quickly received and consistent. Two such systems are now described. The first forms part of a PhD, the second called Ceilidh, which has now become a fully fledged product, was initially developed at Nottingham University.

Automatic Assessment of Shell Programs

[47, Salmon]

The main attributes needed for a system used to assess students work were identified as: rapid feedback to students, appropriate and detailed feedback, and an effective grading system that provides an accurate overall grade as well as information that

identifies the students' weak areas. The system focuses on the software quality factors of correctness (dynamic) and maintainability.

Dynamic Correctness: the program is run against a set of inputs and known outputs to ensure correct behaviour. This is akin to black box testing, which tests the program against the program's specifications without knowledge of the inner working of the program.

Maintainability: the text of the program is assessed for typographical style and complexity to determine its maintainability. Six factors are recognised to measure program style: characters per line, spaces per line, blank lines, comments, indentation, and identifier lengths.

Complexity: the set of factors used is that defined by Zuse, with the inclusion of Halstead's measures. These eleven measures depend upon the construction of a flowgraph (a representation of the control flow of a program). Using graph theory principles on the flowgraph, the following measures are taken: lines of code (Conte et al); number of decisions (nodes) (DEC Zuse); length (Halstead); vocabulary (Halstead); nested level (of nodes) (PEN); the sum of the nested levels (of nodes); Cyclomatic Number (McCabe); number of possible paths (Fenton); number of loops (Hecht); sum of nested levels (of nodes) by loops (NL Howatt); number of nested pairs (Zuse); number of overlapped pairs (Zuse).

The Ceilidh System

[46, Zin & Foxley]

The system focuses on the software quality factors of correctness, maintainability and efficiency. The efficiency of the program is measured simply by timing the

program when it executes, and is only used when specifically requested by the lecturer, as it is not a very accurate metric. Correctness is assessed using 2 methods: static analysis and dynamic testing. Firstly, static analysis is used to identify potential error conditions such as infinite loops, unreachable statements, conflicting conditions, improper nested loops and unused variables. Dynamic testing is used to uncover execution errors in the programs, in much the same way as Salmon's system, by automated black box testing.

Maintainability: The Ceilidh system measures the maintainability of a program by measuring its understand-ability, which is assumed to be inversely proportional to its complexity.

Three methods of calculating program complexity are used: Halstead's software science [M. Halstead] (where the number of operators and operands within a function are counted), McCabe's Cyclomatic number [McCabe] (which maps graph theory onto the line segments of code); and a method that is sensitive to the decomposition of the program into procedures and functions [S Henry, D Kafura].

Program style is also measured using Rees' Ten factors [Rees], which include average line length, comments, indentation, identifier length, use of label and gotos, blank lines, embedded spaces, modularity, variety of reserved words, and variety of identifier names.

Summary

Although hugely different in scale, these two systems both use the same approach to solve the problem of automatic assessment of source code, which is to use

established software engineering metrics to quantify different aspects of the code and then use a weighting mechanism to issue a conclusion or final mark.

Although this is the best approach that may be taken for assessment, such metrics are only useful when providing advice, such as "function x is too complex". They are, however extremely unhelpful when it comes to changing function x to make it less complicated; although advice could be given directly to the student, it would simply be a directive on which area they were weak in, leaving the onus on them to obtain the knowledge needed to overcome their weaknesses.

It is apparent from these systems that for a system to aid the programmer it either needs to give a good explanation / justification of its recommendations or good examples or solutions upon which the programmer can build or adapt.

2.3.3 Conclusion

The routine and well-structured aspects of the coding process have been effectively automated, the most obvious example being the compiler which translates a high-level language into machine instructions. However, the less easily formalised tasks, such as programming, debugging and testing, are still left to the programmer to do. Although currently available static analysis tools go some way to detecting bugs, AI maybe a more appropriate mechanism for attempting to automate this less easily formalised task [35, Walenstein] and for advising on good programming practice.

The main requirements for the system, in the context of these others are:

- It is not adequate to simply collect metrics. Assistance is needed in the form of detailed explanations / justifications of recommendations or the provision of good examples / solutions.
- Flexibility is needed between projects. Rules need to be flexible as some practices result in stylistically correct, maintainable code, but others may provide a more suitable solution.
- The system should not be able to modify the code in any way. It should be offline. Advice should be available only upon request by the user.

2.4 Specification of Requirements (System aims)

The system is required to aid the engineer in the following areas of quality: coding guidelines, good design, program correctness and implied knowledge. These are now described.

Many establishments have a skeletal set of coding guidelines that programmers are expected to adhere to. However, these guidelines are usually vague and incomplete to allow the software engineers a degree of freedom and creativity. Most cover the rules that are undisputed and easy to enforce. They provide a baseline for best practice, but their lack of clarity and completeness about both the programming language used and target platform render them ineffective in defect prevention.

Closely linked to these guidelines is the concept of "good design". Many establishments echo ideas and principles that have been developed within academia as to what represents a good design. Although this system will not be applied to the requirements, analysis and design stages of the life cycle, a reasonable amount of

design is still left to the programmer. Principles such as modularity, information hiding, abstraction, cohesion and coupling can all be measured to some degree at this stage of implementation.

Program correctness is obviously important to software quality. There are two main issues to consider. Firstly, errors concerned with constructs, data structures, iterations and conditional statements, such as buffer overruns. Formal Methods can deal with these issues; the use of clear and concise mathematical proofs to establish correctness of code. However, these are not appropriate for real-life scenarios where tight time-scales apply. The second issue is of the implied correctness; these are errors such as race conditions between threads, which are not easily (if at all) resolved using formal methods. Formal methods give a mathematical and logical framework by which requirements can be refined into design and again into implementation. Industry has adopted the technique mainly in the area of safety critical software. However, formal methods do have limitations; they cannot guarantee that the top-level specification is what was intended [23, Butler & Johnson], the notational difficulty and lack of integration between methods hinders their applicability, and most tools lack industrial strength and are therefore impractical when used on real projects [24, ERCIM].

A variant of implied correctness is target platform specific implied knowledge; a system should be able to enforce rules about the target platform. This aspect of the system would be helpful in detecting errors caused by programmers who are unfamiliar with a new environment. This is important as many errors are related to environment specific facets, such as the failure to close a file handle after a successful file open within the Microsoft Win32 environment.

Reusability and maintenance are also factors that contribute to software quality. There are many existing practices that can be applied to ensure code is easy to re-use and maintain. An establishment's guidelines probably already cover a number of the major ones, as do the principles that are applied for good design. There are also a number of Software Metrics that can be used to quantify the maintainability of code.

For each of the aims listed previously, metrics can be applied to measure the number of defects within a piece of code. However, it is not sufficient merely to return a measure of quality, but also to ensure good quality in the practices applied by the programmer. To do this the system needs to be able to suggest alternative ways to avoid defects.

To be able to suggest alternative solutions the system would need an initial, complete and correct knowledge base of examples, problems and solutions. This is infeasible for a number of reasons, not the least of which is that it is not technically viable. The knowledge within the system must also allow for change.

The "knowledge engineering bottleneck" is the term used by McCorduck to describe the major obstacle to the widespread use of expert systems. The "bottleneck" is the cost and difficulty of building expert systems using traditional knowledge acquisition techniques [25, McCorduck]. Maintaining and refining such a knowledge base would be equally difficult.

This requirement for change reflects the way an organisation's guidelines evolve over time, taking into account new environments, influences and ideas, and changes in the level of quality required. For example, a guideline has been recently introduced to ensure that the checking of a function's pointer parameters is done in a

consistent manner; all pointer parameters in an 'externed' function should be checked. Also, when a new 'bug' has been identified (e.g. in the Windows API, a call to an internal function of Windows actually zeros 12 bytes more memory than it is given, resulting in a stack corruption), the issue must be adopted by the reviewers with immediate 'high' priority. Therefore a learning system is required to allow for the necessary changes in the knowledge.

In a team environment the system should adopt a distributed architecture. By allowing suggestions to be made between team members, the number and diversity of the solutions will be increased. If recommendations are to be made between team members the system will need to be flexible about aesthetic style. The parts of a programmer's style that are not covered by the guidelines will need to be respected. If a team member with a different style makes a recommendation the layout should be changed before it is presented to the other team member. It has been proved that code laid out in the programmer's style is more understandable to that programmer [44, Pomberger].

The system will need to be aware of differing levels of competence within a team environment. Ideally such a system should learn from an experienced programmer and teach or guide new team members. Therefore, the system will adapt a stereotype approach to users, classifying them into categories that represent the quality of their coding. This will allow the system to learn more effectively and efficiently. However, it is important that this mechanism should not appear to be offensive to individual users, especially newcomers [45, Perrolle].

The system must be usable and acceptable to users and workable within the current programming environment. From discussions with engineers at HMGCC it appears that an *easily-integrated demand-driven system* is best suited to their environment, i.e. it should wait until prompted before giving its opinion and not annoy like Microsoft's Office assistant.

Only code that a programmer believes to be ready for system review should be submitted and therefore it is a reasonable assumption that the code will have been compiled and the compiler identified errors and warnings fixed. Potential users were certain that a system that comments on "unfinished code" would not be popular.

2.4.1 Usage

Figure 2 shows the way in which a code review system could be used to minimize the number of defects found in code, prior to the traditional peer code review. The numbers indicate the flow of information. The author submits the compilable source code to the system for review (1), from which findings are reported back to the author for action (2). The fixed code is then resubmitted to the system (3), this allows the system to ascertain the user's preferences, by identifying which findings were actioned and which were ignored. The source code is then submitted to peer code review (4). The peer code review recommendations are fed back to the author (5), the necessary alterations are made and the code is again submitted to the system (6). This allows the system to recognize new classes of bug as identified by the peer code review process, enabling it to include this new information and discover more bugs in future reviews.

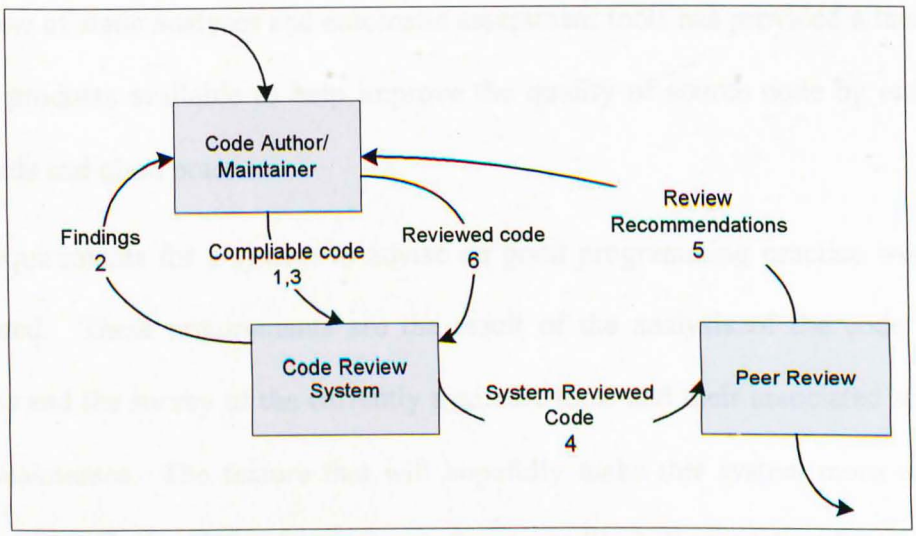


Figure 2: Life cycle including Code Review System.

The envisaged system comprises a set of learning systems, which adapt to their users and represent the users' practices and preferences in a team environment. It is hoped that this system will exhibit similar strengths to the peer code review format, for instance, by improving quality across the team, lowering the number of defects in code prior to peer code review, and by evolving its knowledge of guidelines and solutions.

2.5 Summary

In this chapter, the benefits of peer code review have been analysed and compared to those of software Inspection. The ability of code review to improve the programming skills and knowledge of engineers is complimentary to the ability of software Inspection to improve the process of catching errors in software development. Fewer bugs are introduced by the engineers and more of those that are are found earlier in the development lifecycle, reducing the cost of rework time.

A review of static analyses and automatic assessment tools has provided a landscape of the products available to help improve the quality of source code by enforcing standards and good practice.

The requirements for a system to advise on good programming practice were then presented. These requirements are the result of the analysis of the code review process and the survey of the currently available tools and their associated strengths and weaknesses. The feature that will hopefully make this system more effective than static analysis tools at improving software quality is that it takes direction from the peer code review process, allowing for the team and its dynamics by being adaptive.

These requirements are now analysed and a solution identified to achieve the main aim of providing a team aware system that captures and advises on best practice.

Chapter 3

Analysis & Solution Identification

The requirements of the system as presented in the last chapter are now refined and analysed, to identify the best way to implement a system for advising good programming practice.

In order to capture best practice it is necessary to provide a system which is flexible enough to support the dynamics of the team's standards (defects and solutions) and yet effective enough that this current standard is applied to the code produced. Two top-level requirements are identified as essential for the system to be successful in capturing the benefits of peer code review. It must be adaptive and distributed.

Possible solutions are discussed, with the best suited one identified as agents. A justification to this decision is given by presenting evidence from the agent research

community and by giving a review of a number of projects which use autonomous adaptive agents to assist the user.

3.1 Analysis

To allow the system to mimic the team it needs to comprise multiple nodes, which represent each team member (engineer). Each node is a standalone system that works for an engineer.

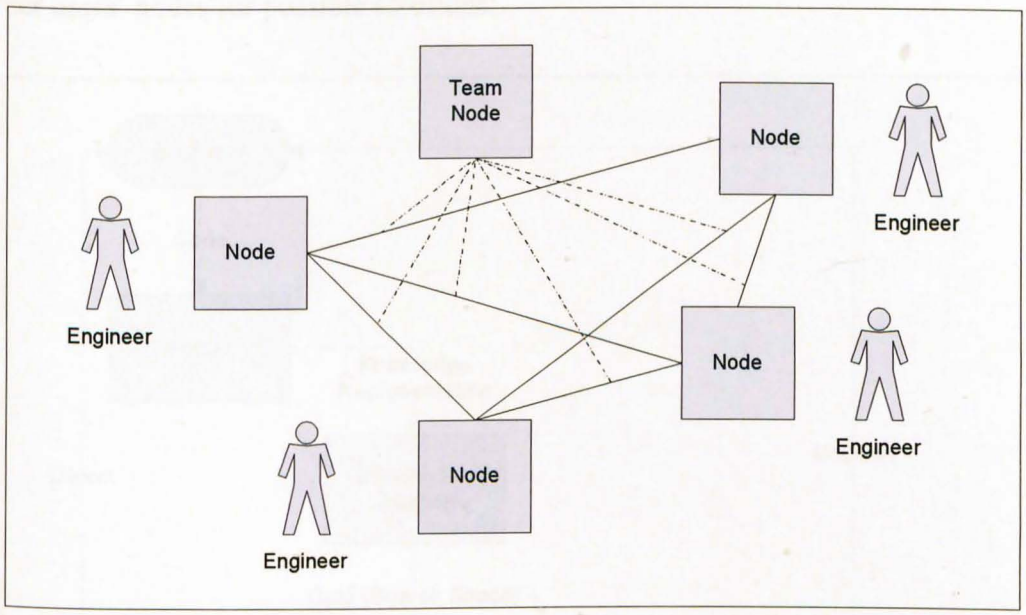


Figure 3: Team Architecture

Figure 3 shows the distributed nature of the team. The system should encompass all engineers who are working to the same set of procedures, standards and constraints. Communication between these nodes is freeform. The team node is included here to symbolise that data about how the team shares ideas (communications between nodes) is captured.

Figure 4 shows an outline of the data flow for a user node. Source code is read into the system, checked that it is syntactically correct (parser) and translated into an

appropriate representation (knowledge representation). This representation is then used to extract basic information about the code (information collation), to give a framework upon which defect detection can be facilitated. Interrogating this extracted information and comparing it against the set of guidelines enables anomalies in the code to be detected. Solutions can then be suggested as workarounds or fixes for the identified anomalies (defects). Solutions are either provided from the system's local knowledge (guidelines and/or history) or by asking other users' nodes for possible solutions.

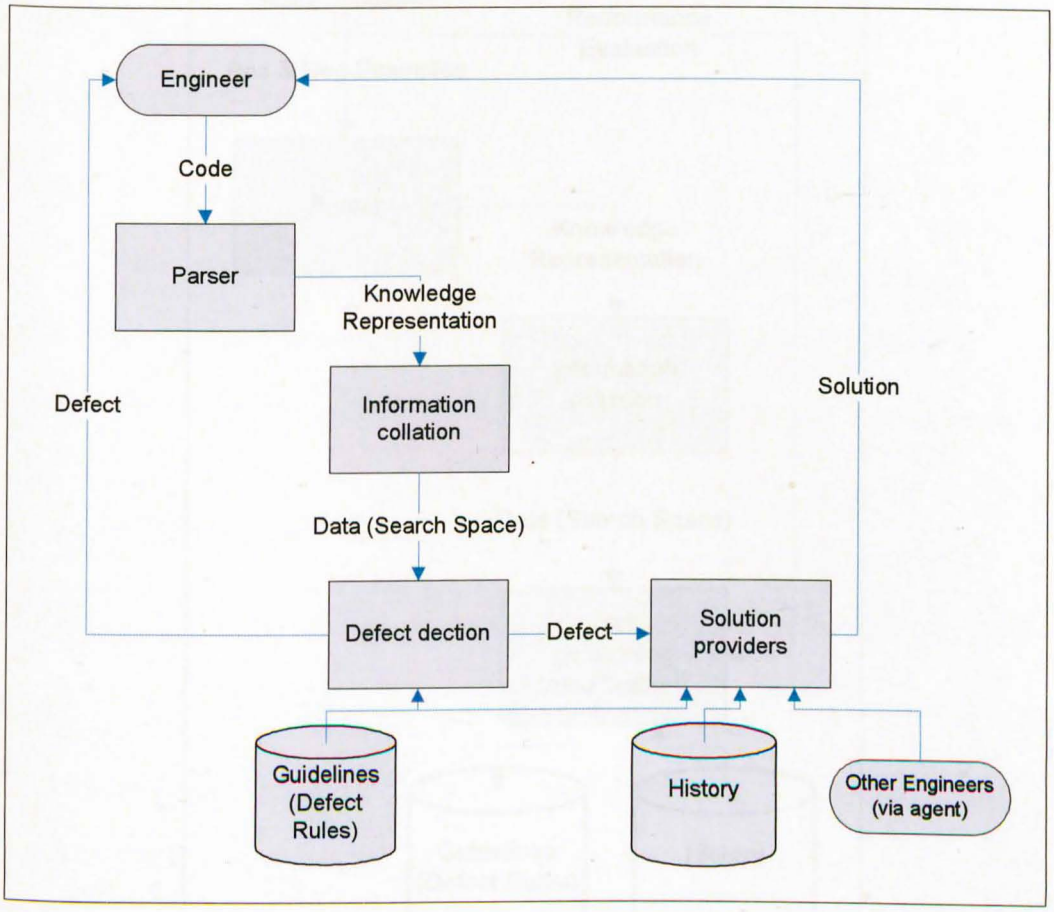


Figure 4: Data flow within a node when detecting defects

Learning by example enables each node to acquire new guidelines and solutions from the user. By supplying positive and negative examples, the user gives the

system a classified learning set. From this set, a generalization (i.e. a guideline or rule that describes good or bad behaviour) can be induced. This enables the system to detect similar behaviour in future examples. Direct feedback is given by the engineer, thus allowing him/her to identify missed or erroneously reported defects back to the system, reinforcing and / or refining its knowledge. The data flow for the learning process is shown in Figure 5.

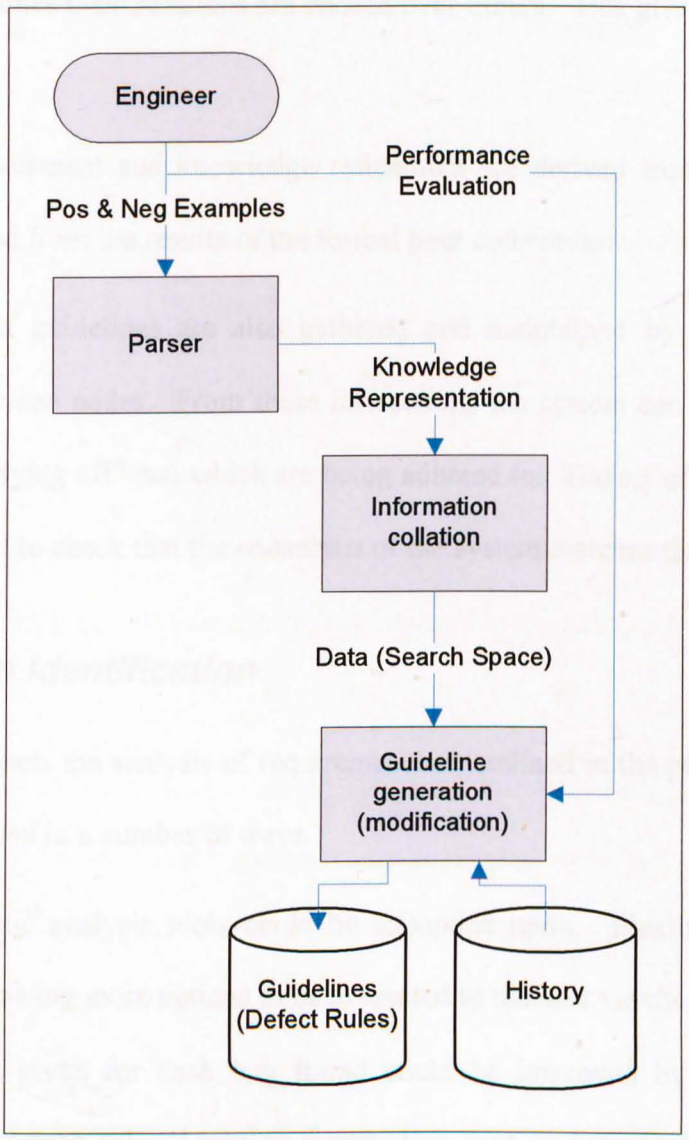


Figure 5: Data flow when learning within a node.

As nodes share solutions and guidelines, and each node records information about the originator, an internal view of the team is developed. An initial view of the team is given using stereotypes, where each engineer is categorised according to their competence. This rating has an effect on the solutions offered and the weighting of confidence in those solutions, as agents prefer solutions from other agents who represent "experts". Over time, the ratings of these engineers will be influenced by the number of times their solutions are chosen over others. This gives a global view of the team.

Learning reinforcement and knowledge refinement are derived from the feedback from the user and from the results of the formal peer code review.

A central set of guidelines are also gathered and maintained by monitoring all interactions between nodes. From these interactions the system can identify which guidelines are "dying off" and which are being adhered to. The set of guidelines can then be reviewed to check that the consensus of the system matches that of the team.

3.2 Solution Identification

A system that meets the analysis of requirements as outlined in the previous section could be developed in a number of ways.

The existing static analysis tools could be expanded upon. Flexibility could be improved, by allowing more options to be presented to the user via the user interface. The information given for each bug found could be improved by incorporating example fixes for each rule. Links to information such as Microsoft's knowledge base could be included in new target specific rules.

This would allow the system greater flexibility and increased usability, as the information supplied would be more helpful to the programmer. But it would not allow for the dynamic flexibility needed, and would not support the team environment.

An expert system could be employed to capture the knowledge of best practice, using knowledge acquisition techniques to extract the fragments of knowledge from the experts. Experienced engineers could also be used as experts to describe their best practice from which rules could be established. Into these rules information about when certain rules should be applied could be captured, allowing the system to be flexible as to which recommendations were made for certain projects or programmers. Solutions could also be incorporated into the system so that they could be presented to the user for defects found.

Blackboard systems could be employed to allow a number of dispersed expert systems to communicate. This would allow each engineer to have their own expert system, and it could communicate with other users' systems as shown in the usage diagram.

The major issue with this design is that the system's knowledge would remain static. To evolve the standards, more expert knowledge would be required. The system would be too static to be able to keep up with current trends and new knowledge. Also, who would be the expert? As identified in Chapter 2, the peer code review process works based on the dynamic consensus of the group, which has so far proven undefinable.

Software agents have proven themselves in a number of capabilities that are required for this system. The two main areas within agency which are relevant are groupware and personal (autonomous) assistants. Groupware affords the system a distributed nature and team awareness. Personal assistants allow delegation, autonomy and adaptivity to be introduced to the system. A brief discussion follows on these fields of research and how such ideas will benefit this system.

3.3 *Why Agents?*

3.3.1 Groupware

Groupware is a term used to describe multi-agent systems that provide support for teams and work groups during the processing of common, relatively unstructured tasks [62, Meterns et al]. Agent-based applications in particular provide support in this area for the information retrieval and decision making within the problem solution process. They permit the supply and administration of important information and content for all members of the group and the acceptance of routine work within this group [63, Maes]. The applications in the groupware area are currently characterized by a limited intelligence and normally consist of several cooperating agents [39, Brenner, Zarnekow & Wittig]. There are a number of multi-agent systems used in team based / computer supported cooperative working, the most notable are PLEIADES [40, Carnegie Mellon University] and more recently RETSINA [41, Carnegie Mellon University].

As previously described the system comprises a set of user-nodes. Each programmer in a team has their own node that adapts to their style and preferences. As these

systems will be working concurrently, co-operation between them is essential to the effectiveness of each node and the system as a whole.

3.3.2 Emergent behaviour

It has been proposed that multi-agent systems can exhibit emergent behaviour [32, Luger & Stubblefield]; "the specification of the behaviour of the agent alone does not explain the functionality that is displayed when the agent is operating" [33, Maes]. It is thought that this behavioural phenomenon occurs as a consequence of the interaction between the components in a multi-agent system. Pattie Maes goes further to say that "the functionality of an agent is viewed as an emergent property of the intensive interaction of the system with its dynamic environment".

If these nodes are modelled as agents, it is logical to assume that allowing them to co-operate using speech-act based communication and interact with the user may allow the system to gain the same benefits as the peer code review, where the whole is worth more than the sum of its parts.

To encourage emergent behaviour the agents should be allowed to communicate freely to achieve their tasks. However, to limit the risks from emergent behaviour, the way in which the agents communicate should be controlled by limiting the performatives they use and by giving each agent a clearly defined role and set of objectives.

3.3.3 Adaptive

"Autonomous agents" have been proved to be useful in applications where adaptivity is required. The user is engaged in a cooperative process in which human and

computer agents initiate communication, monitor events and perform tasks. Over time, the agent or personal assistant becomes gradually more effective as it learns the user's habits and preferences [38, Maes]. The ability to learn represents one of the most important criteria for intelligent operation [39, Brenner, Zarnekow & Wittig]. However, some consider learning in a multi-agent system to be a risk [42, Malone, Lai & Grant], as it is feasible for a system to infer incorrect rules (or fail to infer correct rules).

In the use of software agents as "personal assistants", machine learning techniques are used to get the agent to "program itself". By giving the agent a minimal set of domain knowledge, the personal assistant can learn appropriate behaviour. There are two particular conditions that have to be fulfilled; (1) the use of the application has to involve a substantial amount of repetitive behaviour (within the actions of one user or among users), and (2) this repetitive behaviour is potentially different for different users. This equates well to the process of coding because there is a limited number of constructs and operators within a computer language, and yet individuals combine these elements in different ways (programmers' styles) to achieve the same goal.

3.3.4 Personal Assistant

In section 3.4.1 a number of systems are reviewed which have been developed using agents as personal assistants. These have shown that, if the issue of trust in delegation is addressed with an effective learning mechanism, a useful and effective level of assistance can be attained.

In the author's opinion, it is the ability to independently complete a task that defines an agent, as opposed to it being just a piece of software. More traditional software applications such as Microsoft Word are tools, controlled by direct manipulation. An agent is more than a tool; it is an application to which a task can be delegated. The code review system should review the user's code upon the simple request of "go", then it should seek solutions and present this information back to the user, with little or no help or direction.

3.4 Previous Work: Agents

3.4.1 Personalised agents (Autonomous Adaptive Agents)

A review of several agent-based systems that exhibit learning or adaptivity is now presented.

Letizia

[48, Lieberman]

Letizia is a user interface agent that assists a user browsing the World Wide Web. As the human is surfing (in the left hand pane of the user interface), the agent also surfs, and continuously shows its advice to the user in the right hand pane. Letizia follows the links from the current page and forms an assessment about which page the user should visit next, based on the user's preferences. The model of the user is built from observations, which links the user follows, and what keywords he/she performs searches on.

Newt

[53, Sheth]

Newt is a collection of information filtering interface agents used to filter net-news articles for relevance. The agents are intelligent (using relevance feedback and a generic algorithm) and autonomous. Cartoon characters are used to describe the state of the agents giving a friendly and accessible interface to users of differing abilities.

One interesting aspect of this system is its ability to adapt to the users requirements over time, by using genetic algorithms. However, it is important to note that there is a direct relation between the relevance feedback and the accuracy of the system.

Another aspect of interest is the learning mechanism used. The user is able to provide positive or negative feedback for articles retrieved (akin to defects found) and by providing examples of articles that the agent did not retrieve (positives). This is an example of programming by demonstration.

MAGI

[54, Payne & Edwards]

MAGI is an agent that aides a user in sorting incoming electronic mail. In essence the system is an apprentice which autonomously observes and analyses user behaviour in dealing with email. By observing the user's habits within the e-mail system's user interface, the system can build a profile of the user - a set of rules or weighted metrics that can be used to predict user behaviour with respect to filing incoming messages. Trust thresholds are used to determine how much confidence

the agent has in its predictions. When confident, the agent executes its predictions by moving the emails to the appropriate mail-folder, and feedback is given by the user if the agent is in error. A browser is provided to allow the user to explore the agent's predictions before actions are performed.

One interesting aspect of this system is that one of the learning algorithms used was chosen because it generates human comprehensible rules by performing induction over training examples containing specific features.

Another aspect of interest is that, as the characteristics of the different message types varied, so did the performance of the learning algorithms. One performed better on a certain type of message, but was out-performed by the other algorithm on all the other categories.

3.5 Discussion

In summary, the role of the code review system, as specified in section 2.4, is to advise on good programming practice, not to directly effect changes to the code. Advising allows an avenue of control without being burdensome and enables the user to benefit from the agent's work while retaining responsibility for decisions; advice becomes the communication channel. The user gives advice by describing their desires for the system, and giving a critique of the agent's performance. The agent then advises the user on the course of action to take [48, Lieberman]. Further to those discussed, there are a number of agent-based systems used for giving (and receiving) advice: [49, Morris & Maglio][50, Shearin & Lieberman][51, Kumar et al].

The routine and well-structured aspects of the coding process have been effectively automated, for example, the compiler is able to take the programming language and accurately translate it into machine code. However, less support is available for the more subjective aspects of developing software, such as coding. This is due to the difficult and personal nature of implementing a design. A system which assists the programmer by giving advice would be ideal, as the author would still have freedom and responsibility.

The agent philosophy lends itself to the role of an assistant where advice is given and received, aiding the programmer in the task of detecting and rectifying defects in the code.

The ability to adapt to the user's preferences, by being flexible enough to cope with different projects with differing levels of complexity and on different target platforms, and the ability to match up solutions to defects, will allow this system to be more effective than current static analysis techniques.

This chapter has presented an analysis of the requirements that were defined in the previous chapter. It has detailed the functionality (i.e. adaptivity and communication) that is assumed necessary to capture the benefits of peer code review. Other tools simply try to assert others' coding standards on a single user. There appears to be no system at present that consistently provides fixes or workarounds for any bugs identified.

The system will be adaptive, allowing new guidelines and solutions to be supplied by the user. It will also allow the user to review his/her code using the system, so that defects are identified before peer code review. To do this, it will need to capture

what is looked for during peer code review. In turn, to do this, the system as a whole will need to consist of a number of user systems so that it can learn the traits and preferences of other members in the team.

A discussion on why an agent approach is most appropriate has concluded that an agent-based architecture better achieves the capabilities of delegation, autonomy and social awareness.

Also presented was an overview of some agent systems, which provide assistance to the user by identifying his or her preferences or habits, and then pre-empting them to save the user time. Although it is not desirable for agents within this system to act on any recommendations they make, it is necessary that they do identify, without much direction, the user's preferences for coding.

Chapter 4

System Architecture

This chapter presents the initial and then the refined system architecture, discusses the divergence, and the motivation for the changes. The suitability of the architecture to the problem domain is discussed, and the prototype agent system used to prove the foundational assumptions of the system is described.

This section presents an innovative approach taken in the construction of the agents, which provide a platform for powerful autonomy. Decisions were taken to ensure that the communication between agents is unbounded; that every agent has the ability to communicate with any other agent at any time. To manage this and ensure that the agents remain focused on their goals, each agent has a role (or set of goals) defined within their knowledge. These low level design decisions encourage the benefits of emergent behaviour, whilst reducing the risks from such phenomena.

4.1 Agent Infrastructure

Middleware is a term used to describe the software layer that resides between the underlying operating system and the application layer. Middleware is the standards, models and architectures which exist to promote the use of agents by providing a framework within which the top level functionality of the agent can be specified by the developer, without having to be aware of the lower level issues. Jade, Jack, OOA and Lost Wax provide a common set of programming interfaces that developers can use to create distributed systems. They support communication, security and management capabilities for an agent based system.

However, just as the ability for the system to portray emergent behaviour is necessary, so the ability to allow the agents free communication, but then also restrict this by the assignment of responsibilities and roles will also be needed. This implies a less generalised model, which is cumbersome and contains much redundancy.

4.2 Agent Architecture

Agent architecture is the fundamental engine underlying the autonomous components that support effective behaviour in real-world dynamic and open environments [52, Luck et al]. There are three main models of an agent: reactive, deliberative, and hybrid. Reactive or behavioural agents operate in an event-response manner. Deliberative agents reason about their actions; a common model for a deliberative agent is the BDI model (beliefs, desires and intentions). Hybrid agents try to combine the best of both approaches, choosing which approach is more beneficial at any moment in time.

There are a number of toolkits around to aid the development of each of these models, especially for the Hybrid architecture: Jack from AOS and Disciple from Tecuci, are examples.

In addition to the implicit benefits afforded to agent architectures (commonly referred to as Agent-Oriented Design), the proposed agent architecture has a key advantage. It allows the freedom to include different capabilities and functionality in different agents, whilst allowing them to communicate in a consistent way. There is no significant overhead, as normally incurred by more generic, all encompassing architectures. Due to the use of Java Expert System Shell (JESS), the agents are easy to configure and the roles easy to define.

4.3 Proposed System architecture

As discussed earlier, an agent approach is appropriate for the sharing of knowledge between individual users. It is also appropriate for the knowledge sharing and co-operation within each node. Figure 6 shows the proposed system architecture [65, Mercer & Greenwood].

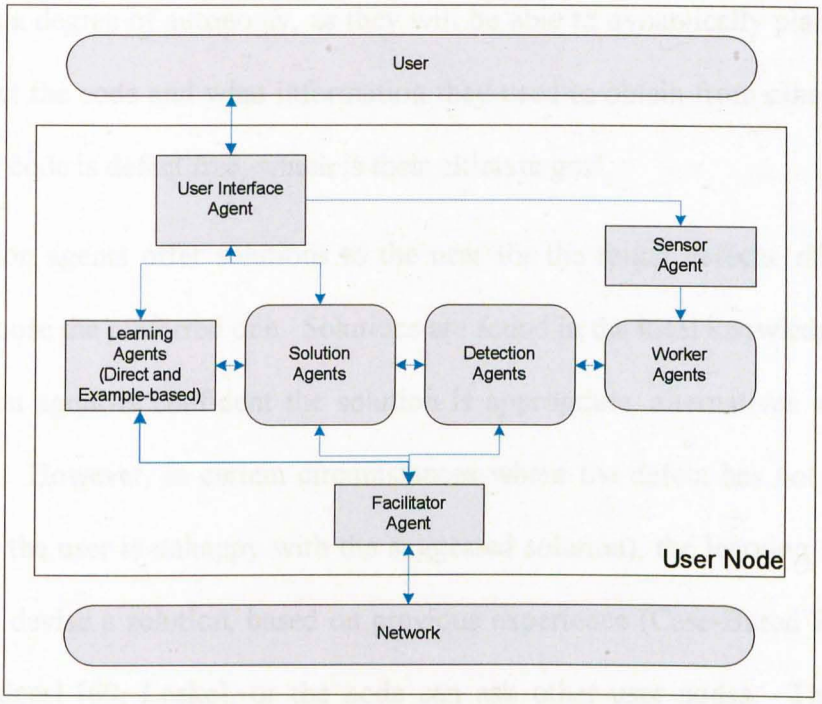


Figure 6: Proposed system architecture

The sensor agent is responsible for parsing the source code and producing the language specific representation of the constructs within the code. The worker agents then perform rudimentary reasoning over the constructs to extract information about the basic elements - declarations, function definitions and statements.

The detection agents interrogate the worker agents to ascertain information about the code's structure, control flow and so forth. They use this to detect anomalies in the code. The detection agents co-operate to ensure all of the guidelines have been checked. To do this they employ a BDI (Beliefs, Desires and Intentions) architecture to assist them in attempting to model the intentions of other agents [66, Finin]. The BDI model allows the agents to employ practical reasoning (reasoning directed towards actions [67, Wooldridge & Parsons]), by analysing the current environment, and their own knowledge (beliefs) and goals (or desires). The agents decide what to do next and what they need to do to achieve their plans (intentions). This will allow

the agents a degree of autonomy, as they will be able to dynamically plan how they will inspect the code and what information they need to obtain from other agents to ensure the code is defect free, which is their ultimate goal.

The solution agents offer solutions to the user for the found defects, allowing the user to choose the preferred one. Solutions are found in the local knowledge base. If the solution agent is confident the solution is appropriate, alternatives will not be suggested. However, in certain circumstances where the defect has not been seen before (or the user is unhappy with the suggested solution), the learning agents can attempt to devise a solution, based on previous experience (Case-Based Reasoning) [68, Kolodner] [69, Leake], or the node can ask other user nodes. The learning agents can also directly learn from the user (direct-learning), allowing the user to identify new defects or solutions that the system can immediately integrate into its knowledge.

All interaction with the user is accomplished through the interface agent, which allows pertinent information about choices to be reported back, not only to the solution agents, but also to the learning agents.

As shown in Figure 2, when the user has made the appropriate alterations the code is entered into the system again. This allows the learning agents to adapt their knowledge based on solutions that have been implemented and upheld by the user, defects that have not been rectified, and solutions that have been ignored.

If no further defects are found, the code can then be submitted for peer code review. The results of the peer code review allow the system to refine and reinforce its knowledge. The revised code, which includes the peer code review

recommendations is again entered into the system. The system can again adapt, resulting in either, a) a new or amended guideline where new defects have been spotted, b) an amended guideline where a known defect has been ignored, or c) a solution being modified or replaced where suggestions have been overturned.

At any point during this lifecycle the user can add new guidelines and provide new solutions, but it is intended that the system does not allow the user to edit or remove them. This will ensure the independent evolution of the system, as negative or less preferred solutions will *die out* over time without user intervention.

By monitoring the user selection process and the outcome of peer code reviews, the system is able to make judgements as to the user's level of competence. It will also be able to build a model of the user's view of the team. As this information has bearing on the solutions provided and the learning mechanism of the system, each user will initially be assigned to a stereotype group, which reflects approximately their level of competence.

By monitoring interaction between the team members a model can be built of the team. A consensus of guidelines can be also be constructed that should reflect the team's current level of quality.

4.4 Revised System architecture

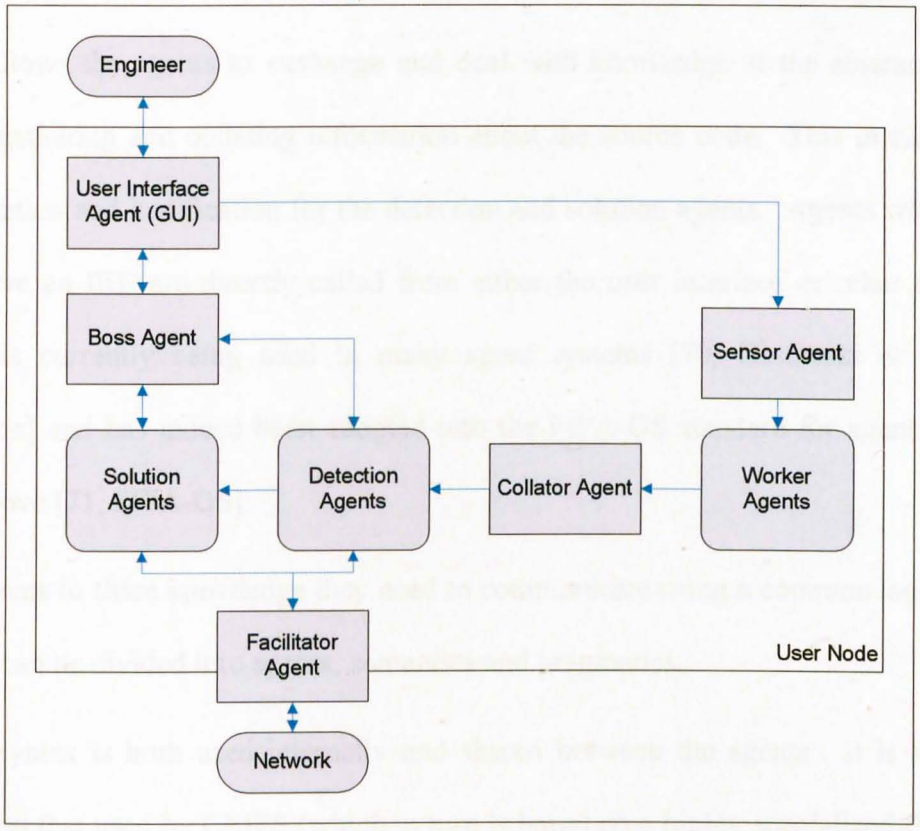


Figure 7: Revised system architecture

Figure 7, shows the revised system architecture and denotes the number of agents and the predominate information flow between them. All agents within this system are constructed in the same way; they have the same execution patterns and communication capabilities, (described in the following section: Agent Anatomy). The different roles are also described in the following sections, together with an overview of each agent's capabilities.

4.4.1 Agent Anatomy

The prototype system is implemented in Java. As the agents reside on the same platform, they run in their own threads, allowing pseudo concurrent execution.

The worker, sensor and collator agents include an inference engine, Java Expert System Shell (JESS) [64, Friedman-Hill], as their internal reasoning engine (IRE). This allows the agents to exchange and deal with knowledge at the abstract level when gathering and collating information about the source code. This in turn aids explanation and justification for the detection and solution agents. Agents which do not have an IRE are directly called from either the user interface or other agents. JESS is currently being used in many agent systems [70, Ossowski et al][72, Kennion] and has indeed been adopted into the FIPA-OS standard for agents 1.3.0 and above [71, FIPA-OS].

For agents to share knowledge they need to communicate using a common language, which can be divided into syntax, semantics and pragmatics.

JESS syntax is both used internally and shared between the agents. It is loosely based on that used by CLIPS (which in turn is based on a highly specialized form of LISP) and is highly expressive. The content of a message is a JESS fact. When processing incoming messages, the contents (facts) are asserted into the IRE.

The semantics are described in the ontology of the system, which defines a common vocabulary. It describes the domain, including representations for common source code elements (declarations, functions, control flow), system specific details (how to feedback to user, etc) and knowledge sharing axioms (wants-to-know, need-to, etc).

The pragmatics are based on KQML [73, Finin & Labrou], which describes the way the agents communicate, e.g. how to ask a question (ask, ask-all), how to respond to a question (reply), etc. These processes are hard coded into the agent, as the reaction to such requests as an "ask" is predetermined.

The basic structure of an agent is given in Figure 8. On initialisation all agents register with the facilitator. The facilitator is the only other agent that an agent knows about when first created. The member function "Run" is the thread's procedure; on exiting this function the thread is destroyed.

```
Initialise()
        Register with facilitator
        Initialise IRE

Process()
        Create thread

Run()
        While (agent is active)
        {
                Assert received messages into IRE
                Run IRE
                Sleep (400 ms)
        }
```

Figure 8: Pseudo code - basic structure of an agent

When "Process" is called (by the interface agent, upon creation) the agent creates its main thread this is done here so that the agents start processing when all agents are created and registered with the facilitator. The thread then executes the "Run" function; this ensures that the agent processes new data (messages) and then sleeps, allowing other agents to execute. This is repeated continually whilst the agent is active (the main-loop). "Run IRE" means invoke the rule base engine to process the newly asserted data (facts). This may result in the agent completing a task, in the agent requesting more information (sending a message) or in no action if the agent is waiting for more data to continue.

When an agent wants to send a message, it uses the facilitator's message event handler. Event-handlers in Java are functions which allow manipulation of data

structures within one agent to be done by another agent's thread. Each agent has a message event handler. The facilitator's message event handler is used to forward the message, by calling the message event handler on the recipient agent. Other agents' message event handlers simply add the newly received message to their message queues. The process of sending a message is shown in Figure 9. The function agent uses the message handler in the facilitator to find the message handler in the expression agent. It then adds the message to the expression agent's message queue, ready for the expression agent to process when its main loop wakes.

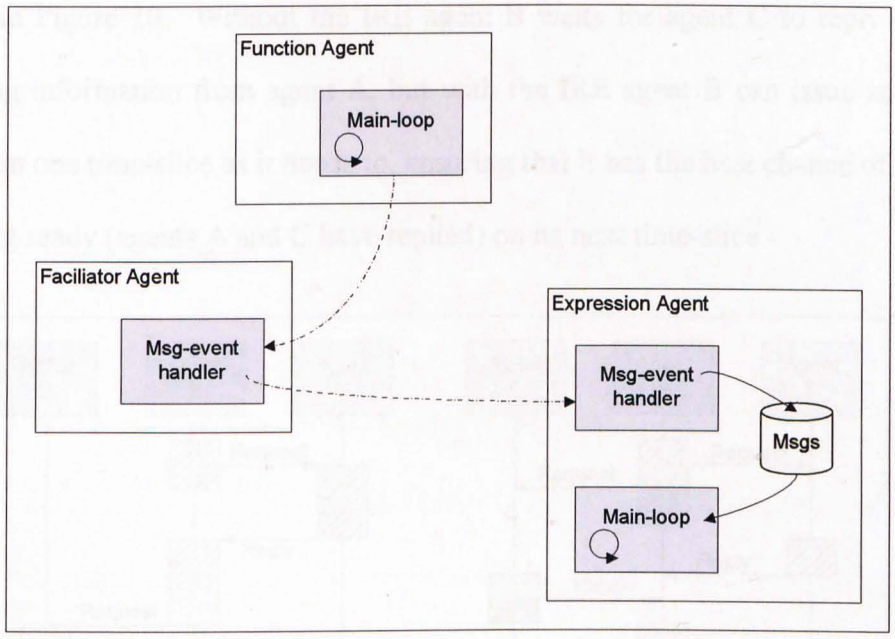


Figure 9: Mechanics of agent communication

The advantage of using a rule-based engine in this way is that programming is easier, goals can be specified easily, and processing remains at the knowledge level. It also gives the agents a degree of autonomy, as the almost random nature of the operating system's time-slicing and IRE processing determines which goals are completed first.

This uncertainty in the order of processing can cause problems, such as agents answering questions before they have enough information (for example, when asked for all variable declarations, the variable agent was replying, "don't have any", before it had received them). Sub-goals were devised to specify which requests agents could service, given their current knowledge level.

The content field contains a JESS fact; this allows the content to be directly asserted into the agent's IRE. Without IREs the agents would be too procedural and concentrate on one task (goal) at a time. They would also be more reactive, as can be seen in Figure 10. Without the IRE agent B waits for agent C to reply before requesting information from agent A, but with the IRE agent B can issue as many requests in one time-slice as it needs to, ensuring that it has the best chance of all the data being ready (agents A and C have replied) on its next time-slice.

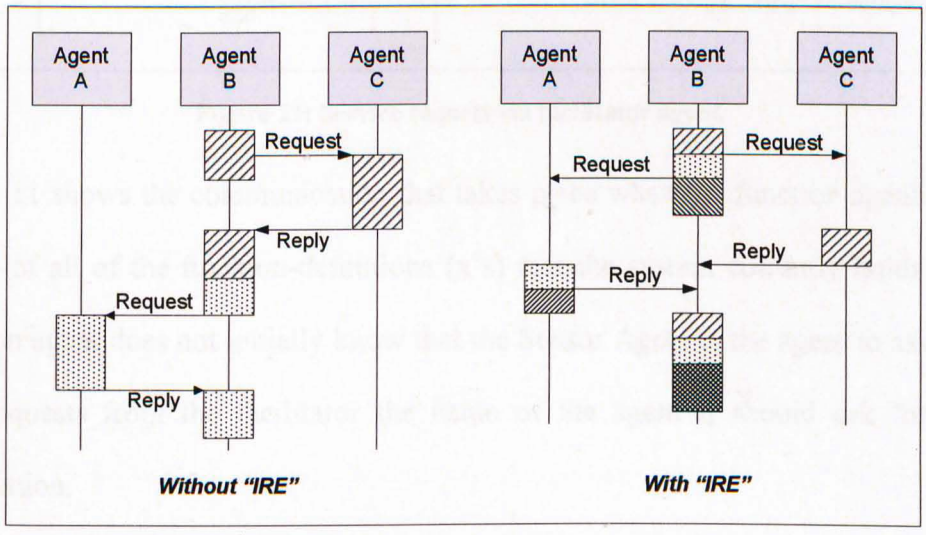


Figure 10: Processing flows with and without IRE

Figure 11 shows the contents of the KQML messages for the "ask" and "reply" performatives. As can be seen, the sender and recipient are described, as is the type of message (the performative) and the content. The reply-with and in-reply-to fields

allow the originating agent to keep track of what it asked (i.e. what the question was for this answer).

4.4.2 Facilitator Agent

The facilitator agent acts as a broker agent, matching agents to service-requests from other agents [74, Genesereth].

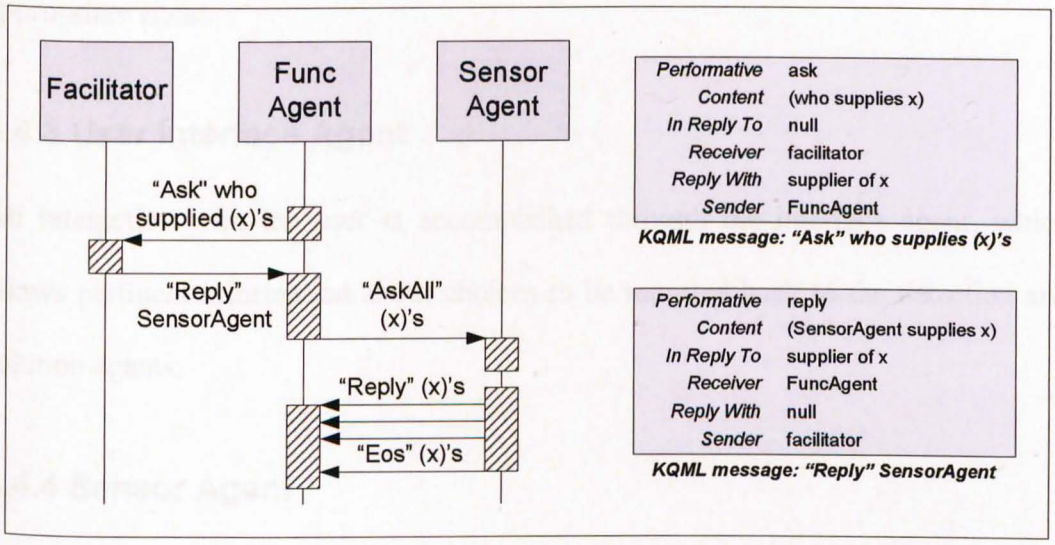


Figure 11: Service request via facilitator agent.

Figure 11 shows the communication that takes place when the function agent wants details of all of the function-definitions (x's) that the system currently holds. The function agent does not initially know that the Sensor Agent is the agent to ask, so it first requests from the facilitator the name of the agent it should ask for such information.

As specified in KQML there are four possible ways in which the services can be brokered by facilitator agents [75, Finin, Labrou & Mayfield]. This system adopts the "recommend" model, which requires the requestor to ask the facilitator for the name of the agent which supplies the information / service it requires. The requestor

then deals direct with the recommended agent. This dealing direct method stops the facilitator from being swamped with requests which are not necessary, as all the agents are on the local node. Another mode of service brokering is for the facilitator to become the 'middle man', passing on requests and responses between two parties. This model is preferred for network based interactions, as internal details of the node do not need to be known. The facilitator can direct incoming requests to the most appropriate agent.

4.4.3 User Interface Agent

All interaction with the user is accomplished through the interface agent, which allows pertinent information about choices to be reported back to the detection and solution agents.

4.4.4 Sensor Agent

The sensor agent is responsible for parsing the source code and producing the language specific representation of the constructs within the code. A representation of the code is needed so that a meaning can be given to each construct to enable further processing / reasoning to be realised. The function of any reasoning scheme is to capture the essential features of a problem domain and make that information accessible to a problem solving procedure. Abstraction, the representation only of information needed for a given purpose, is an essential tool for managing complexity. Expressiveness and efficiency must also be balanced to ensure the representation is efficient, but care must be taken not to limit the representation's ability to capture essential problem-solving knowledge [76, Luger & Stubblefield].

To produce a representation of the source code, the sensor agent has to parse the input in a similar manner to a compiler. This is done in five stages:

1. Comments are removed from the code.
2. Pre-processor directives are removed from the code. A rudimentary attempt to translate #defines is made. However, this is only done once and therefore defines which contain defines are not resolved.
3. The code is then tokenized - broken into the basic building blocks of the language: operators, punctuators, constants, keywords and identifiers. All white-space is removed.
4. The tokens are then divided into sets that represent constructs within the code.
5. The final step is to construct a representation (semantic tree) for each construct.

One beneficial side effect of this is that the system is now able to display the code to the user in a more structured way, giving confidence that the source code has been interpreted successfully and providing a basis upon which defects can be reported and feedback given. A screen shot of this is given in Figure 12, which was captured from the prototype system as described in section 4.5.

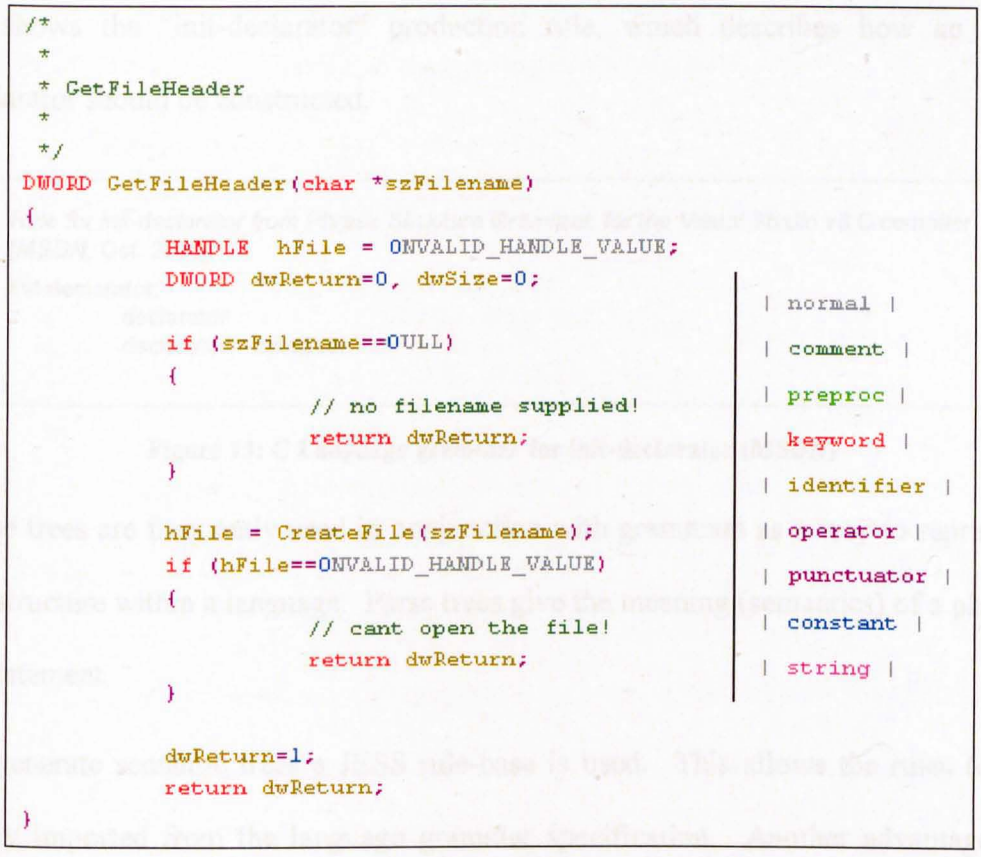


Figure 12: Syntax highlighting by the Sensor Agent, with Key

White space is "thrown away" by the system at this stage. As such, guidelines and rules relating to style cannot be covered by this system. It was decided that rules covering style issues were redundant, as the advance of Integrated Development Environments (IDEs) means that changing the style of code can be done easily. Borland's J Builder v9 allows the programmer to specify their style in a number of dialog boxes, the IDE then reformats and presents code to the programmer in that style on-the-fly. Therefore, the requirement to accommodate style is no longer relevant, as every programmer can view code to their own preferences.

To construct a semantic tree, the language specification is used. The language specification defines the syntax of the language. There are 69 production rules defined in the C Language grammar specification, as defined by the MSDN. Figure

13 shows the "init-declarator" production rule, which describes how an init-declarator should be constructed.

Rule for init-declarator from Phrase Structure Grammar, for the Visual Studio v6 C compiler [MSDN, Oct. 2000].

```
init-declarator:
        declarator
        declarator = initializer
```

Figure 13: C Language grammar for init-declarator (MSDN)

Parse trees are frequently used in conjunction with grammars as a way to represent the structure within a language. Parse trees give the meaning (semantics) of a phrase or statement.

To generate semantic trees a JESS rule-base is used. This allows the rules to be easily imported from the language grammar specification. Another advantage of including a rule base is that it allows the system to work at the knowledge level and not, as with compilers, in a non-human readable form. This allows for easier explanation and justification within the system as a whole, as outlined in section 4.5.2.

The C Language grammar specification as defined by the MSDN, without the Microsoft specific extensions, translates into 222 JESS Rules. Figure 14 shows the JESS rule for the init-declarator rule as shown in Figure 13.

Rule for init-declarator as defined in the SensorAgent's C Language rule-base.

```
(defrule init-declarator_declarator
 ?c <- ( node (from ?f) (to ?t) (value declarator ) )
 =>
 ( assert ( node (from ?f) (to ?t) (value init-declarator) (children (create$ ?child)) ) )
 )

(defrule init-declarator_declarator_equals_initializer
 ?c1 <- ( node (from ?f1) (to ?t1) (value declarator ) )
 ?c2 <- ( node (from ?f2) (to ?t2) (value equals ) )
 ?c3 <- ( node (from ?f3) (to ?t3) (value initializer ) )
 ( test (eq ?f2 (+ ?t1 1) ) )
 ( test (eq ?f3 (+ ?t2 1) ) )
 =>
 ( assert ( node (from ?f1) (to ?t3) (value init-declarator) (children (create$ ?c1 ?c2 ?c3)) ) )
 )
```

Figure 14: Jess rule for init-declarator

When fired, the rule-base attempts to generate a tree starting from the leaf nodes (or terminals). If successful, all nodes will be contained within a tree, whose root is that of a 'translation unit'. This is the term used to describe a unit equivalent to a single source code module (.c or .h file).

Stage 4 is achieved by applying a heuristic algorithm to the set of tokens. Although the agent is capable of constructing a single semantic tree for the source code module, the inclusion of the heuristic ensures performance is acceptable by dividing the tokens into smaller, more manageable sets. When the heuristic is not applied, all tokens in a source code module (.c file) are mapped to a single semantic tree, the root of which is a 'translation-unit'. When the heuristic is applied, all tokens map to a set of semantic trees. These trees can either represent a function-definition, declaration or statement. Figure 15 shows the way in which the heuristic divides the tokens up into constructs; note the empty compound statements in constructs 0, 3 and 6. The contents of a compound statement are not needed to determine if the parent construct is valid, therefore they are removed and processed individually.

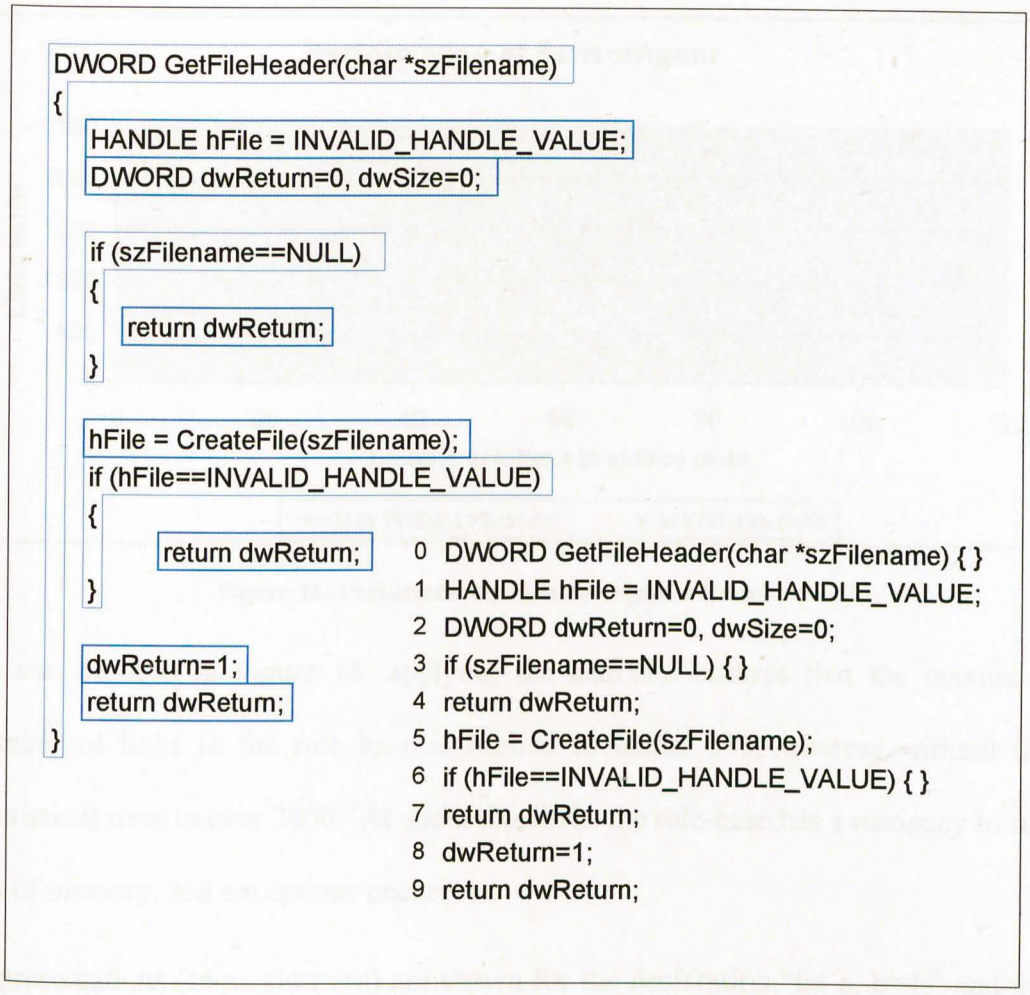


Figure 15: Construct identification using Sensor-Agent's Heuristic

The heuristic ensures that only tokens related to a single construct are used when generating a semantic tree. This limits the amount of processing needed to ensure the syntax is correct and to generate the representation to an amount that is reasonable for the user and the operating system.

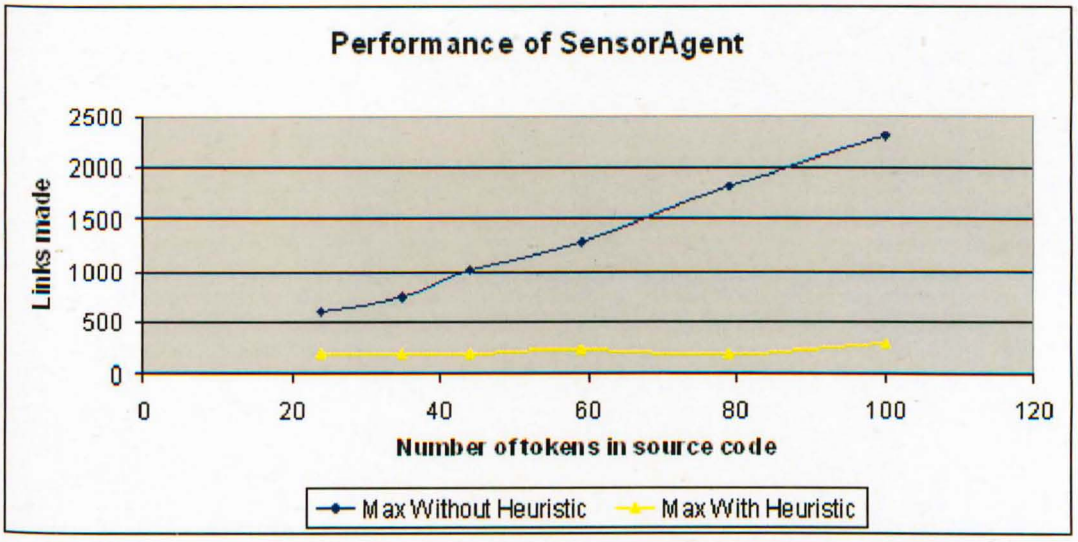
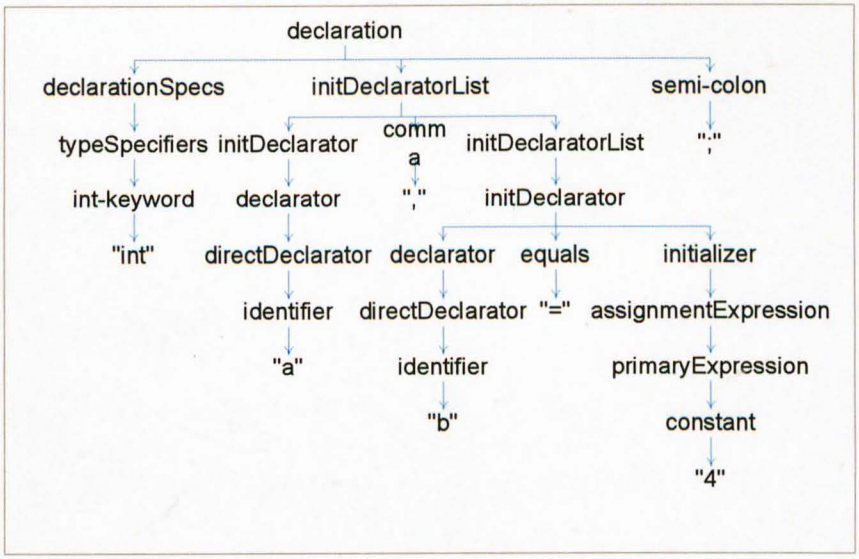


Figure 16: Performance of Sensor-Agent's Heuristic

As can be seen in Figure 16, applying the heuristic ensures that the maximum number of links in the rule base is limited to below 500, whereas without the heuristic it rises to over 2000. At about this limit, the rule-base has a tendency to run out of memory, and exceptions occur.

Representations (semantic trees) are shown for the declaration "int a, b=4;" and the statement "a=a+1;".



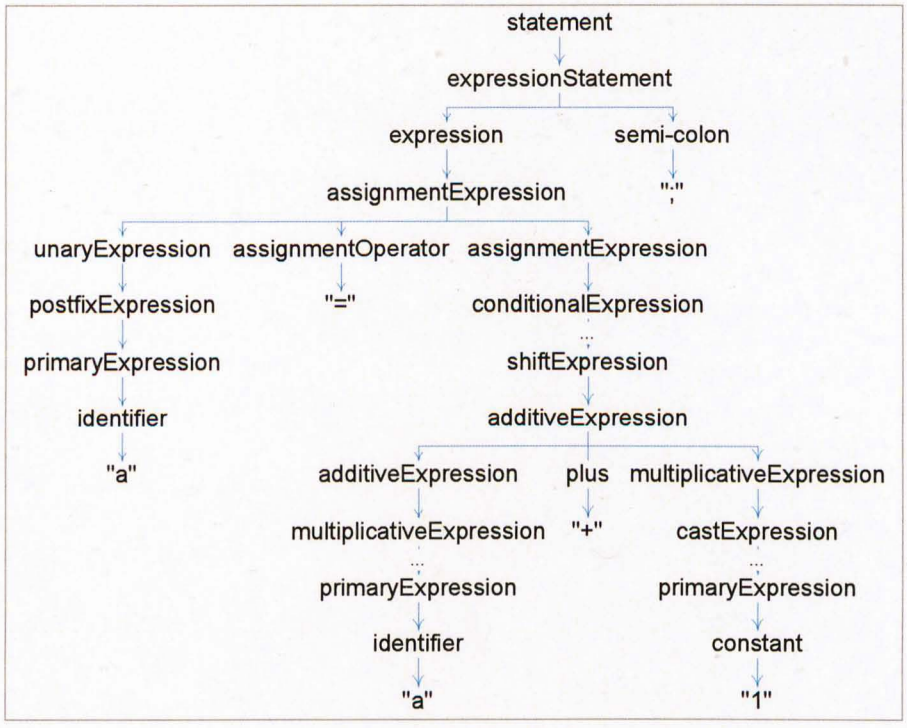


Figure 17: Semantic trees, as constructed by Sensor Agent.

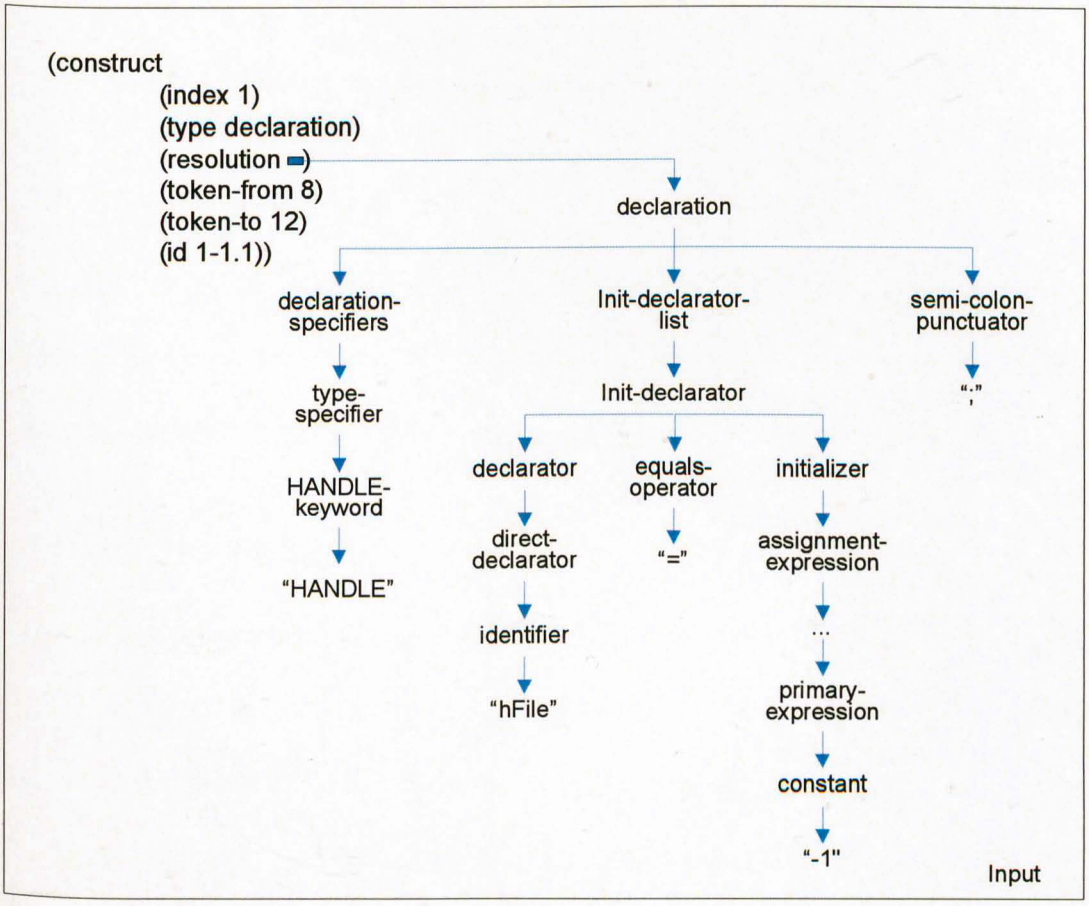
These representations allow the agents to interpret the meaning of the constructs. For example, in the first syntax tree, the ordered set of sub-clauses {expression-statement, assignment-expression and additive-expression} imply that the expression is the assignment of an addition. In the second tree, the leaf nodes of branches which contain "directDeclarator" are the names of the variables declared. These semantic trees give the ontology of the statements - an implied meaning to the statements that the system can rely on.

4.4.5 Variable Agent

The variable agent is responsible for collating information about variable declarations: global, stack (function-scope) and parameters.

To do this the variable agent receives from the expression agent (described in section 4.4.7), semantic trees and construct information about all the declarations within the source code. Information about all the parameter declarations of functions is also received from the function agent.

Using the semantic trees of each declaration, the variable agent is able to extract information about the variable, such as its type, name, initial value, etc., as shown in Figure 18. The variable agent needs to request information from other agents to gain certain information, such as the scope of the variable (from the function agent) and what actually happens to a variable in a function (from the expression agent).



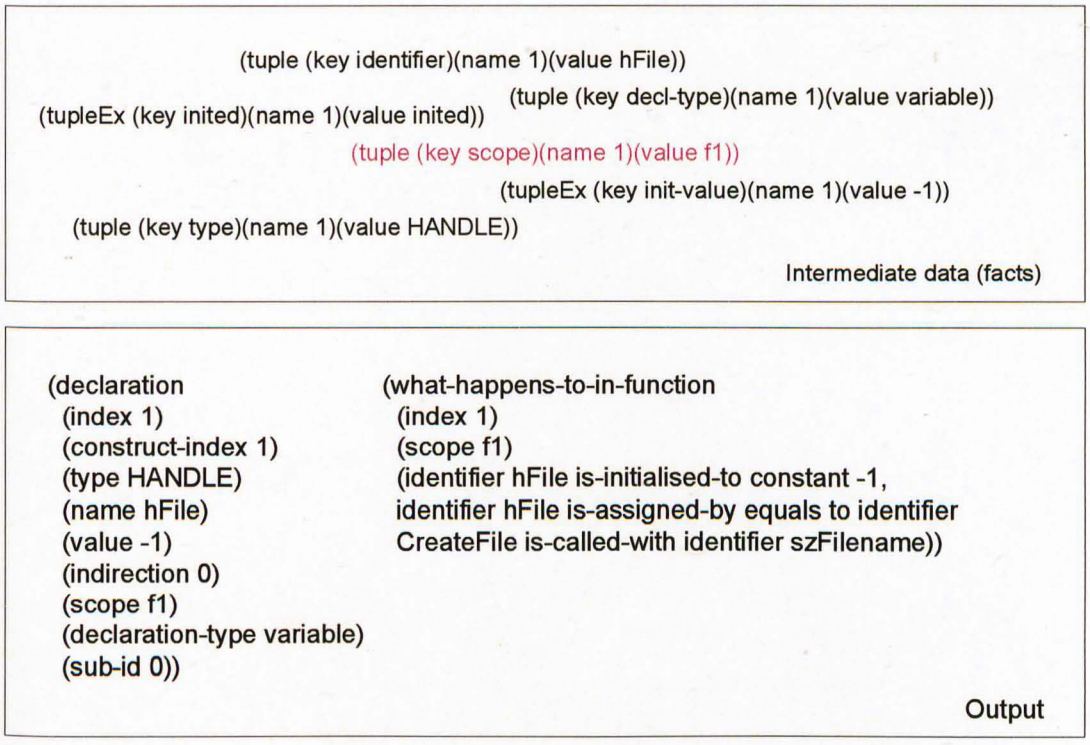


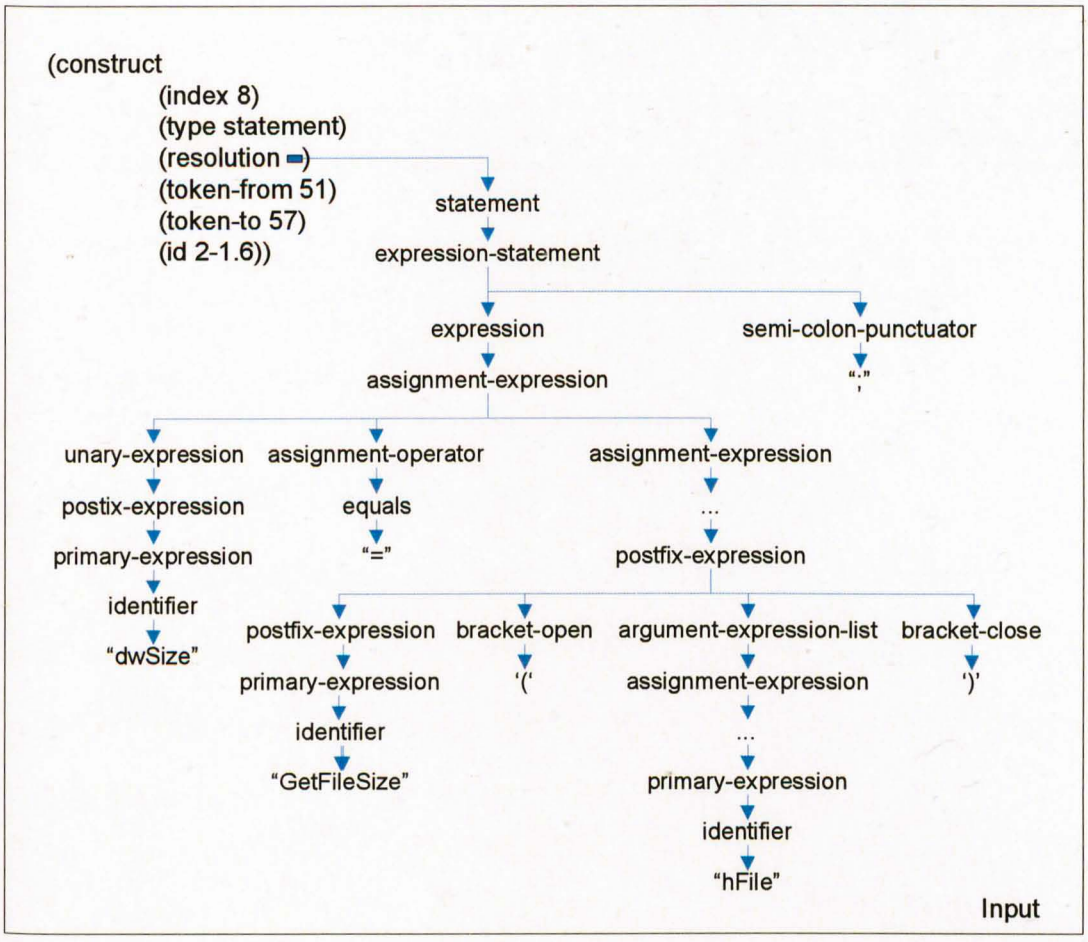
Figure 18: Processing of a variable declaration in JESS

4.4.6 Function Agent

The function agent is responsible for collecting information about function-definitions. To do this, the agent receives information from the expression agent. The semantic trees are used to determine the features of the function definitions.

4.4.7 Expression Agent

The expression agent receives all of the construct information from the sensor agent. Its first responsibility is to forward the declarations and function-definitions to the variable and function agents respectively. The expression agent is then able to process the remaining constructs to ascertain the behaviour of the source code. It translates the statements into facts as shown in Figure 19.



(expression-at (name 8) (index f2) (value identifier dwSize is-assigned-by equals to identifier GetFileSize is-called-with identifier hFile))

(variable-reference (name GetFileSize) (index 8) (value identifier GetFileSize is-called-with identifier hFile))

(variable-reference (name dwSize) (index 8) (value identifier dwSize is-assigned-by equals to identifier GetFileSize is-called-with identifier hFile))

(variable-reference (name hFile) (index 8) (value identifier hFile is-used-as-argument))

Intermediate data (facts)

```
(variable-usage (name GetFileSize) (index f2) (value identifier GetFileSize is-called-with
        identifier hFile) )
(variable-usage (name hFile) (index f2) (value identifier hFile is-initialised-to constant -1 is-
        declared , identifier hFile is-assigned-by equals to identifier CreateFile is-called-with
        identifier szFilename , identifier hFile is-double-equals-by constant -1 is-tested , identifier
        hFile is-used-as-argument , identifier hFile is-used-as-argument) )
(variable-usage (name dwSize) (index f2) (value identifier dwSize is-initialised-to constant 0 is-
        declared , identifier dwSize is-assigned-by equals to identifier GetFileSize is-called-with
        identifier hFile , identifier dwSize is-not-equals-by constant -1 is-tested , identifier
        dwSize is-assigned-to identifier dwReturn) )
(control-flow (name f2)(index f2)(value "1 2 3  <4 > 5 6  <7 > 8 9  <10 > 11 12") )
                                                                        Output
```

Figure 19: Statement process in JESS (expression agent)

Another responsibility of the expression agent is to ensure the source code has been written/interpreted correctly; i.e. a sanity check. To do this the agent collects the knowledge from the variable and function agents, and displays to the user an interpretation of the source code, as shown in Figure 20.

```
Function: MyGetFileSize:
 Declarations:
 -> declaration(f2, dwSize)
 -> declaration(f2, hFile)
 -> parameter(f2, szFilename)

 Statement Descriptions:
 -> (1, variables-of-type HANDLE are-declared identifier hFile is-initialised-to constant -1)
 -> (2, variables-of-type DWORD are-declared identifier dwSize is-initialised-to constant -1)
 -> (3, identifier hFile is-assigned-by equals to identifier CreateFile is-called .... )
 -> (4, if identifier hFile is-not-equals-by constant -1 then )
 -> (5, identifier dwSize is-assigned-by equals to identifier GetFileSize is-called .... )
 -> (6, identifier dwSize is-returned)

 Control flow:
 -> (f2, 1 2 3 4  <5 > 6)

 Variable References:
 -> dwSize
 ---> (2, identifier dwSize is-initialised-to constant -1 is-declared)
 ---> (5, identifier dwSize is-assigned-by equals to identifier GetFileSize is-called .... )
 ---> (6, identifier dwSize is-returned)
 -> CreateFile
 ---> (3, identifier CreateFile is-called-with identifier szFilename)
 -> szFilename
 ---> (3, identifier szFilename is-used-as-argument)
 -> hFile
 ---> (1, identifier hFile is-initialised-to constant -1 is-declared)
 ---> (3, identifier hFile is-assigned-by equals to identifier CreateFile is-called-with .... )
 ---> (4, identifier hFile is-not-equals-by constant -1 is-tested)
 ---> (5, identifier hFile is-used-as-argument)
 -> GetFileSize
 ---> (5, identifier GetFileSize is-called-with identifier hFile and constant 0)
```

Figure 20: Summary of expression agent's knowledge

This allows the system to identify rudimentary errors, such as the undeclared variable references, etc. It also allows the user to readily identify where the system has failed to correctly interpret the code.

4.4.8 Collator Agent

The collator collects all the pertinent information from the worker agents and passes it to either the detection agents or the evaluation agent, depending on the task at hand. When detecting defects, the collator prepares the information as a set of LearningDataObjects (described below), and passes it to the detection agents. When

the user describes a new guideline by supplying positive and negative examples, the collator forwards the differences as LearningDataObjects to the evaluation agent, so that they can be added to long term memory.

An important data structure used by the defect agents and learning mechanism is the LearningDataObject (LDO). An LDO describes an element of code in a learning space (this is discussed fully in section 5.2). In basic terms, a learning space refers to a set of attributes that describe an element of code; learning spaces represent a variable, function, module or project. The attributes of a learning space are ascribed dynamically (at runtime) to the LDO; a set of attribute names and their associated type and value are stored within the LDO structure. Figure 21 shows an LDO containing information about a variable in the variable learning space.

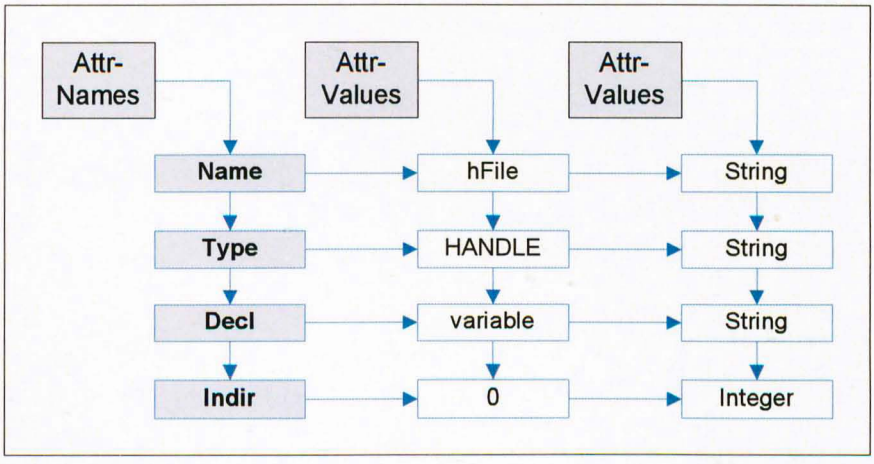


Figure 21: A LearningDataObject_t instance, describing a variable

The flexibility of this class allows LDOs to be easily changed and formatted during the learning process.

4.4.9 Detection Agents

The detection agents interrogate the data collected by the collator agent to ascertain information about the code's structure, control flow and so forth. They use this to detect anomalies in the code. The detection agents compete to ensure as many defects as possible are found with as high as possible accuracy.

It was initially intended that the detection agents would use the BDI model [78, Rao/Georgeff]. However, the model in its entirety is considered to be very complex [79, Brenner] and, for the requirements placed on the detection agents within this system, would prove to be cumbersome and generate an unneeded processing overhead. Deliberative agents are also considered sub-optimal when it comes to learning, as reactive agents are able to adapt quickly to a dynamic environment and receive new information through their many interactions with other objects [80, Maes].

Not becoming entwined in the full BDI model does not mean that the agents do not already contain some benefits of the model. Goals (Desires) are still specified within the IRE (sub-goals which dictate current knowledge level, and therefore requests that can currently be serviced). Planning, or the chaining of tasks to achieve the goal, is also inherent in the rule based structure of the IRE. The antecedents and consequents act inherently as the pre and post conditions of actions. The inference engine chains these rules (tasks) together using the facts (beliefs) to achieve a plan (intentions) that will attain its top-most-goals (desires).

So although these agents are simple in the way that they do not deliberate, do not take great effort to decide what to do next, or reason about each others internal state,

they are appropriately aware of their own state and knowledge such that they are able to function in the community successfully.

The responsibility of the detection agents is two-fold: they not only find the defects in the source code, but they learn how to identify new defects. It was initially intended that separate agents be used to facilitate the learning within this system. However, the task of identifying defects within code supplied is relatively trivial compared to that of learning. Therefore, the two roles were combined into one agent model. The mechanism by which the agents learn will be described in detail in the next chapter. In summary, to learn to detect new defects, the agents use the history (long term memory) of all positive and negative examples, pertinent to the rule to be learnt, such that a generalisation can be induced.

To detect defects, each agent looks for patterns within the set of LearningDataObjects supplied by the collator agent for all rules the agent has previously learnt. It surveys the supplied data reporting the findings, together with a gauge of the agent's own confidence to the evaluation agent.

4.4.10 Evaluation Agent

The evaluation agent is responsible for collating all the defects reported by the detection agents, assessing the system's confidence in the recommendations using weightings (as described later) before reporting them to the user. There are many detection agents looking for defects in the source code, and each agent is looking for contradictions to many guidelines. Due to the indefinite nature of the learning mechanism, some defects may be reported in error. Therefore, a mechanism by which a confidence rating can be calculated is needed, so that only those defects /

recommendations the system is confident about are reported to the user. The evaluation agent uses a number of factors when determining this rating:

- The originating agent's confidence:
    - The confidence the agent has in the rule.
    - The confidence the agent has in the clause of the rule.
- The evaluation agent's confidence:
    - In the agent which reported the defect for this rule.
    - The highest confidence reported by an agent.
    - The number of agents reporting this defect.

The originating agent has no concept of time, and is therefore unable to provide a confidence based on past performance. However the evaluation agent maintains a history and is able to alter its confidence in an agent-rule pairing based on user feedback. This mechanism allows the system to be made aware of which agents are good for which defects.

This agent also manages the history (long term memory) of positive and negative examples - the core data of the system. Not only is the list of all LDOs held by the evaluation agent, but also the list of the rules and details about their positive and negative examples.

As we cannot expect the user to predict the optimum learning space for a guideline, a rule may be induced in many learning spaces, although one learning space will yield predominantly better results. The evaluation agent relates rules to learning spaces. Three sets of LDOs - positive, negative and not-applicable - are described for each

rule and learning space pairing. Figure 22 shows the rule for "No uninitialised variables", where no examples have been provided in the function space, but a number have been created in the variable space.

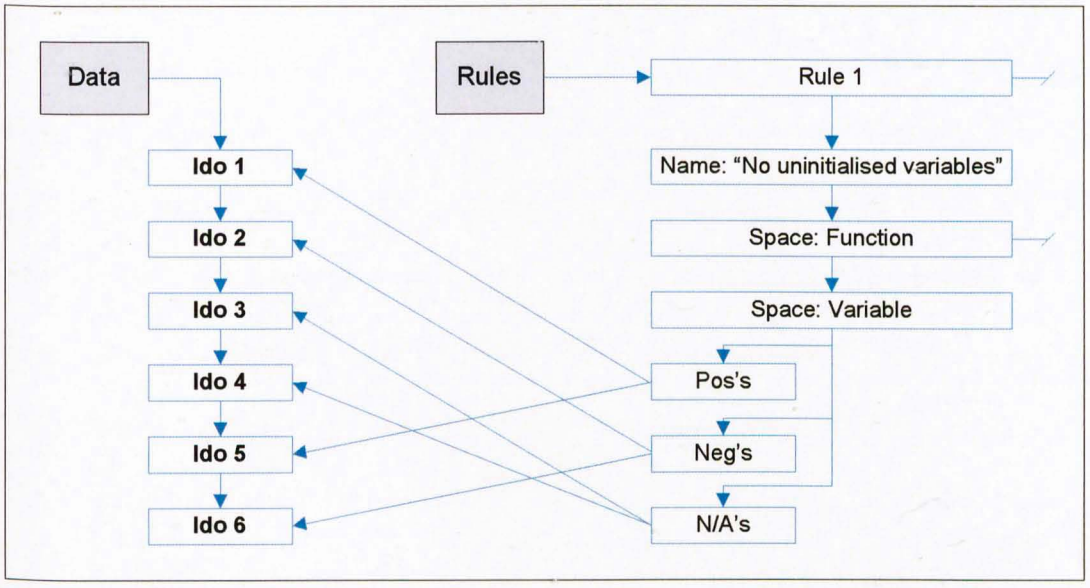


Figure 22: LongTermRule_t instance, describing "No uninitialised variables".

4.4.11 Solution Agent

The solution agent offers solutions to the user for the found defects, allowing the user to choose the preferred one. Solutions are found in the local knowledge base. If the solution agent is confident the solution is appropriate, alternatives will not be suggested. However if the agent is unable to find a solution, or is not confident enough in the current selection, it can ask other users' agents for alternatives.

It was initially proposed that the solution agents would embody a case-based reasoning engine that would enable them to match up solutions for defects from a repository of good examples keyed by the defect itself. However, preliminary investigation has indicated that this is overly complex for what is required, by using

the defect guideline itself and the history of good examples, enough information is available to be of use to the user (see section 5.6).

4.4.12 Knowledge reinforcement & System wide issues

It was originally planned that the system would use the findings of the 'real' peer code review to refine and reinforce its knowledge. This has been superseded by the immediate and direct feedback from the user via the source window. Once a defect has been reported the user is able to approve or disagree with the findings. If the user agrees with the defect the agent's confidence is boosted. If the user disagrees the example is added to the negative list for that rule and the learning algorithm for the agent(s) that erred is retriggered to generalise a new concept.

4.5 The Prototype System

A prototype system has been developed and tested in the real world situation of HMGCC. Certain assumptions have been made: the language to be reviewed is C as defined in the MSDN[1]; the system is limited to detecting defects; the detection guidelines are limited to those that are easily measurable.

4.5.1 Observations

The test plan aimed to prove that the system was comparable to others, in its ability to parse the source code, produce the correct syntax trees and detect defects. For the

[1] Microsoft Developer Network (October 2003), Visual C++, C Reference.

former the prototype system was exposed to a variety of source code modules within the HMGCC environment. The system was able to correctly parse the files and successfully reason over the elements within the code. However, it had some difficulty with some of the Microsoft header files included as part of the Windows API. This was due to the Microsoft-specific extensions to the 'C' language, about which the sensor agent was ignorant.

For defect detection, the testing concentrated on one guideline - 'appropriate use of variables'. The prototype system was compared against LCLint and the Microsoft Compiler[2]. This guideline was chosen as it is easily measurable when comparing performance against other systems.

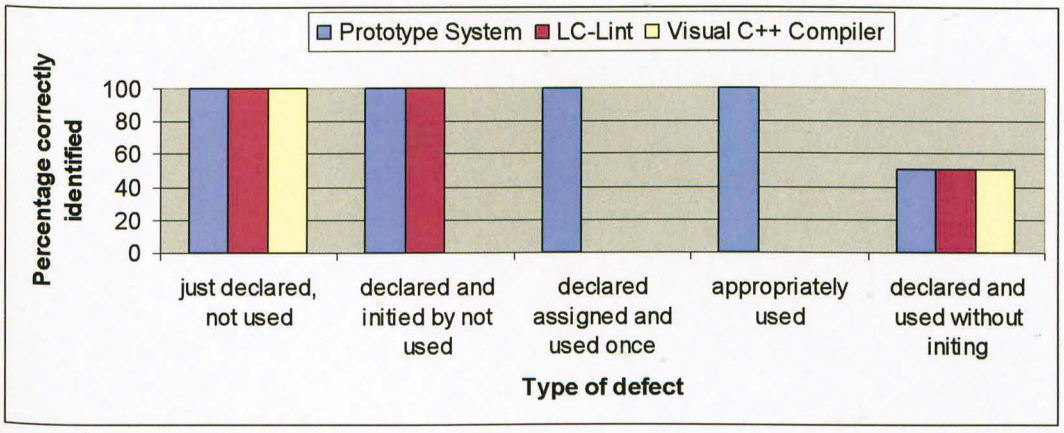


Figure 4: Test results for guideline: "appropriate use of variables".

As can be seen from the results, the prototype system is able to analyse and detect more anomalies in the code than either of the other two systems used. The most notable difference between the systems was that only the prototype was able to

[2] Visual Studio, Visual C++ v6.0 with Service Pack 3 applied, on warning level 3.

recognise variables that should be made constants. This highlights the prototype system's ability to detect anomalies in the code that are not just related to correctness, as recommending that a variable should be made constant is a typical peer code review recommendation to improve the maintainability of the code.

The testing of the first prototype served to demonstrate that it is a good foundation upon which to expand the system. It is able to compete with other static analysis tools when it comes to understanding / parsing the code, and it has the potential to outperform the other tools when it comes to the information gathered prior to defect detection. The defect detection in the prototype was too simple to draw any further conclusions on its performance.

4.5.2 Discussion

Development of the prototype system was aided by the powerful alliance of JESS and the Java language, allowing knowledge and control to be integrated easily. There were, however, a number of drawbacks, mainly in the area of performance. During development it was shown that the system is able to keep the language specifics at the knowledge level only, hence improving the portability of the system, but this proved to be too resource intensive, and limited the size of source code that could be reasoned over at any one time. Language specific heuristics were implemented such that the system was able to fragment the code before processing it at the knowledge level. Although this improved the usability of the system, it detracts from its portability.

A more efficient way of disassembling the code would be to include a parser, or use intermediate compiler data within the sensor agent to describe the code. But this

would be detrimental to other aspects of the system; when defects were reported to the user, a translation would have to be made to ensure that the description of the defect related to the original source code in a human-understandable form. The system would in fact be taking something which is human-readable, translating it into a machine-oriented representation, and then translating it back to human-readable form. It would also have made debugging the system more difficult. Ensuring the system works with knowledge that is human-readable not only makes building and debugging the system easier, but explanation and justification are almost inherent in the learning process (described in the next chapter).

In the homogeneous architecture of the prototype, a strong development benefit emerged in respect of the combination of the JESS rule-base and KQML. In the prototype system message contents were explicitly asserted into the rule base. This allowed the system to implement the ontology directly, as no processing or parsing of information was needed.

4.6 Conclusion

Section 4.3 presented the initially proposed agent architecture. This was refined and extended in 4.4. A prototype system based on the revised architecture was then tested and evaluated. The results of which are promising, showing that the architecture is a good foundation upon which learning can be facilitated. The innovative approach of including multiple agents in one user node and the autonomy afforded by the implementation brings a number of benefits to the developer.

- The ability of the agents to communicate in a non-prescribed way gives easier scalability.

- The self awareness of the agents and their state, coupled with the restriction of the performatives available for use, resolve many of the issues related to freeform communication of autonomous agents.

Chapter 5

Learning

This chapter refines the responsibilities of the collator, detection and evaluation agents by presenting a symbolic-based learning algorithm. It describes how the inputs for the learning algorithm are handled and presents a novel approach to learning, through the use of multiple, relatively simple learning algorithms deployed concurrently to solve the same problem.

The ability to learn must be part of any system that claims to possess general intelligence [81, Luger & Stubblefield]. Luger and Stubblefield go on to say that learning is important for practical applications of artificial intelligence, due to the difficulty in building expert systems using traditional knowledge acquisition techniques. One solution to this "knowledge engineering bottleneck" would be for

programs to begin with a minimal amount of knowledge and learn from examples, high-level advice or their own exploration of the domain.

If we take Herbert Simons' definition of learning, which states that it is "any change in a system that allows it to perform better the second time on repetition of the same task or on another task drawn from the same population", a number of difficulties for learning algorithms present themselves. Firstly, learning from experience means generalising from experience (induction), i.e. the learner must acquire knowledge that will generalise correctly. So the learning algorithm needs to discover which aspects of the experience are most likely to prove effective in the future. Another problem faced by learning algorithms is that change may not be good. The learning algorithm must be able to prevent and detect such problems and overcome the noise of bad examples present.

There are three main models for learning - symbolic, connectionist (neural) and evolutionary (genetic). Symbolic approaches build on assumptions of knowledge based systems; the primary influence on the program's behaviour is its base of explicitly represented domain knowledge. The central aspect of this hypothesis is the use of symbols to refer to objects and relations in a domain.

Connectionist systems de-emphasise the explicit use of symbols in problem solving. They attempt to capture the intelligence that arises in systems of simple, interacting components.

Evolutionary approaches attempt to mimic the process underlying evolution: shaping a population of individuals through the survival of its most fit members.

As the learning domain is that of a computer language, which itself is completely symbolic, it seems appropriate that we remain in a symbolic space for the learning, because it will be the objects (variables, statements, function etc) and how they relate to one another (function calls, assignments, jumps etc) that form the basis of the rules by which a defect can be defined.

Induction (learning a generalization from a set of examples) is one of the most fundamental learning tasks. Concept learning is a typical inductive learning problem. Given examples of some concept, such as "cat", "good-stock investment" or indeed, "buffer overflow", we attempt to infer a definition that will allow the learning algorithm to correctly recognize further instances of that concept. A method of concept learning is needed in this system to facilitate guideline learning with the aim of adapting to find new defects in code.

The first prototype allowed us to assume that we are able to obtain information suitable for basic defect detection from the parsed source code. The next step is to expand the system such that it is able to employ learning techniques so that it can learn the traits of defects and be able to recognise defects in future pieces of code.

As machine learning is applied to real-world tasks, difficulties arise that do not occur in simpler textbook experiments. One such difficulty is selecting the best attributes to use for learning from a large set of candidate attributes. Ideally, a learning algorithm's generalization performance would improve when it is given the information supplied by additional attributes. Unfortunately, the opposite often occurs; additional attributes can interfere with other more useful attributes [82, Caruana & Freitag].

To achieve the level of learning that would be desirable in such a system the following had to be considered. Given the number of attributes that could be extracted from the code, the number of ways each attribute could be interpreted and the number of different defects that the programmer may want the system to identify (concepts learnt), the search space would be too cumbersome for use on conventional hardware and operating systems.

Another complication is the requirement that the learning algorithm should not to be too reliant on the structure / ordering of the training data. As it is intended that the system will always learn and refine its knowledge, it must learn from a real-life perspective – which does not allow tailoring of input. This is directly related to another known difficulty of machine learning, in that the quality and quantity of training data is an important issue for any learning algorithm. Without extensive built-in knowledge of a domain, a learning algorithm can be totally misled attempting to find patterns in noisy, insufficient, or bad data. Therefore, the system needs to be aware of the domain.

One luxury that is afforded this system is that it doesn't have to be 100% accurate, unlike other agent-based learning systems, for example [84, Perugini et al]. The user is ultimately responsible for decisions made. A survey of programmers at HMGCC showed that the accuracy of the system could be relatively low - if it only catches one bug the system would be considered useful. However the tolerance of false-positives was significantly less, as these would lead to wasted time. For testing purposes an accuracy rate of 70% was considered sufficient.

5.1 Model of a learning agent

The need for learning within this system was argued in sections 2.4 and 4.4.9. By incorporating a learning mechanism in the detection agents, they will be able to learn new guidelines and therefore detect new defects. This will allow the system to adapt to the changing coding standards of the team over time.

A model of a learning agent is given in Figure 23. This shows that there are two main components of a learning agent in this system - a learning element and a performance element. The performance element is responsible for taking in the precepts and deciding on appropriate action. The learning element takes knowledge from the performance element, plus feedback on how the performance element is performing, and determines how to make the performance element perform better.

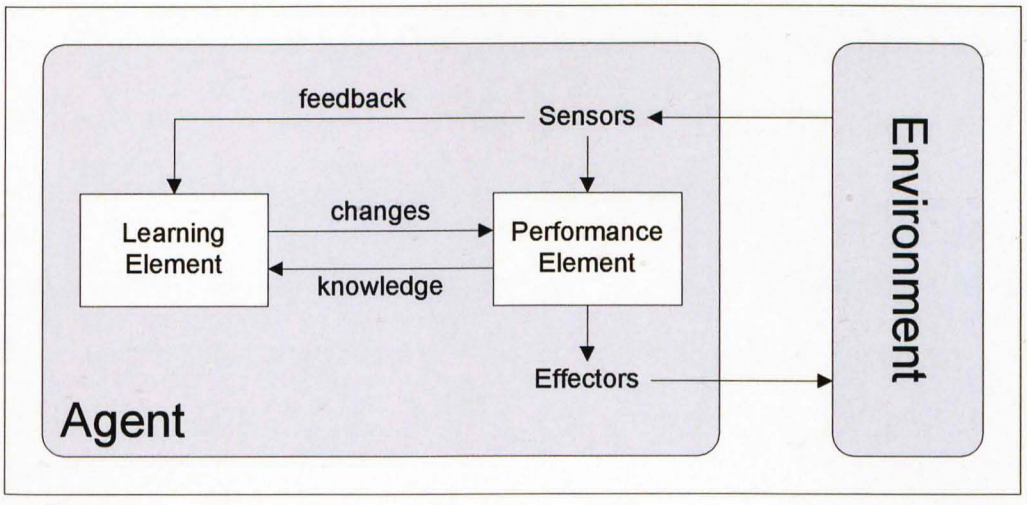


Figure 23: Model of a learning agent

There are a few differences when compared to the general model of learning agents presented in Russell & Norvig [91, Russell & Norvig]. For example, the model proposed does not require a critic (a module used to tell the learning element how well the agent is doing). This is because the user will give feedback to the system if

it incorrectly identifies or misses a bug in the source code. A problem generator (a module which suggests actions which lead to new and informative experiences) would be equally redundant, as it is assumed that enough code will be passed through the system for it to have no need to self generate new knowledge finding exertions.

In this system the user will not be able to describe the guidelines to the system if it gets it wrong. When a learning system only receives some evaluation of its action (in this case, correct or incorrect) it is referred to as reinforcement learning.

5.2 Attribute & Feature selection

To facilitate learning the search space needs to be reduced. This is achieved by attribute and feature selection. Attribute (or variable) selection refers to the problem of selecting input variables that are most predictive of a given outcome. Feature selection refers to the selection of an optimum subset of features derived from these input variables [86, Koller & Sahami].

There are many potential advantages to automating the attribute selection process. It allows the system designer freedom to identify as many useful attributes as possible and new attributes can be introduced more easily. In domains where the world changes, this approach allows the current best attributes to be those best suited to the current state of the world. Lastly, it allows the set of used attributes to change dynamically as the amount of training data changes [82, Caruana & Freitag]. Much research has been, and is being, carried out on the problem of automated attribute / feature selection [87, Raman & Ioerger], [88, Talavera] and [89, Raman and Ioerger].

Although manually selecting attributes is normally considered challenging and can lead to inferior selection, it was not a problem in this case. The selection was achieved via domain knowledge; in a static domain (programming languages are well defined and static) the benefits gained from automatic attribute selection are no longer needed.

Attribute selection in this system is achieved by the use of learning spaces, where the attributes are selected based on domain knowledge. The search space is constrained to two learning spaces: variable and function. A set of attributes that describes a variable and the actions performed on / with that variable is defined as the variable learning space. A set of attributes that describes a function and the statements contained within it is defined as the function learning space. Two further learning spaces could be accommodated - module and project learning spaces - which will allow reasoning at higher scope levels (helpful for rules such as unused functions).

The decision to partition the attributes as such was based on a study of the HMGCC global coding guidelines [90, HMGCC]; the majority of the guidelines were aimed at function/statement or variable level detail (22 out of 35). This was also evident with the syntax rules of PC Lint, where a survey of a subset of rules showed that the number of rules for variable, statement, function, module and project scope were 11, 17, 12, 7 and 2 respectively. Although the investigation showed that more rules relate to statements, learning rules relating to functions were considered more challenging for the system as many rules in this category rely on the interaction between a number of elements: statements, sub-statements, variables and variable-references.

Data is extracted from the worker agents, as shown in Figure 20, and is then organized into a table of attributes as shown in Table 2. However, this data is still not conducive to learning.

Name	Type	Decl	Indir.	Value	Action
hFile	HANDLE	variable	0	-1	identifier hFile is-initialised-to constant -1 is-declared , identifier hFile is-assigned-by equals to identifier CreateFile is-called-with identifier szFilename , identifier hFile is-not-equals-by constant -1 is-tested , identifier hFile is-used-as-argument
dwSize	DWORD	variable	0	-1	identifier dwSize is-initialised-to constant -1 is-declared , identifier dwSize is-assigned-by equals to identifier GetFileSize is-called-with identifier hFile and constant 0 , identifier dwSize is-returned
szFilename	char	parameter	1	nil	identifier szFilename is-used-as-argument

Table 2: data collated from worker agents into "variable" learning space.

Again some domain knowledge is needed to ensure that the attributes presented to the learning algorithm are prepared and formatted to achieve their full potential. Some attributes are made more specific and some more general. Within the variable learning space, the value attribute denotes the value assigned to the variable when it was declared. However, a more general attribute which denotes whether the variable was initialised as a Boolean value may be more effective. A collection of extracted / formatted attributes is referred to as a learning plane.

Domain knowledge leads to the learning planes that are included in the system. Rules about variable naming dictate that a learning plane is needed which allows the system to reason at the character level over the name, as shown in Table 3.

Name	Type	Value	Decl	Indir.	Inited	0	1	2	3	4	5	6	7	8	9
hFile	HANDLE	-1	variable	0	Y	h	F	i	l	e					
dwSize	DWORD	-1	variable	0	Y	d	w	S	i	z	e				
szFilename	char	nil	parameter	1	N	s	z	F	i	l	e	n	a	m	e

Table 3: data formatted in the "variable" learning plane.

To handle rules as described in the testing of the prototype system, a learning plane is needed which generalises the attribute values. Table 4 shows the varEx learning plane, which categorises the actions into either initialised(inited), assigned, tested or used.

Name	Type	Decl	Indir.	Action List
hFile	HANDLE	variable	0	Inited, assigned, tested, used, used
dwSize	DWORD	variable	0	Inited, assigned, used
szFilename	char	parameter	1	used

Table 4: data formatted in the "varEx" learning plane.

This use of learning planes reduces the noise in the data, as only one aspect of the data is supplied at any one time to the learning algorithm. By running the algorithm in each plane, many generalizations will be induced, some being more accurate than others.

5.3 Classification

Due to the requirement that the system is always learning and should be adaptive to its user, it is important that the user has significant influence over its learning.

For learning purposes, the system allows the user to input two different fragments of code that represent positive and negative examples of a rule, as shown in Figure 24. If this is the first time a user has submitted an example of the rule, the system will prompt the user for a description of the rule. This functionality allows the programmer to submit code to the system that was changed by a peer code review, i.e. the before and after. The system is only given the two code fragments, the name of the "rule" it should create or affect, and the solution's (positive example's) origin.

```
a)
  float myFunc(int nCount, long total)
  {
    float percent;
    percent=(float)nCount/(float)total;
    return percent;
  }
b)
  float myFunc(int nCount, int nTotal)
  {
    float fPercent;
    fPercent=(float)nCount/(float)nTotal;
    return fPercent;
  }
```

Figure 24: Negative and positive examples of variable naming

The first task for the system is to identify the differences within the source code fragments; this allows the system to reduce the noise in the data, by excluding information that is identical in both the positive and negative examples. This is achieved using a similarity engine [92, Wetzel]. The knowledge from the collator agent is extracted into the learning space. The variable learning space for the examples presented above is shown in Table 5, where each row represents an LDO.

Name	Type	Value	Decl	Indr	Action	Set
nCount	int	0	variable	0	nCount is-cast-to float is-divided	positive
nTotal	int	0	variable	0	nTotal is-cast-to float is-divided	positive
fPercent	float	nil	variable	0	fPercent is-assigned-by equals to nCount is-cast-to float and nTotal is-cast-to float are-divided , fPercent is-returned	positive
nCount	int	0	variable	0	nCount is-cast-to float is-divided	negative
total	Int	0	variable	0	total is-cast-to float is-divided	negative
percent	float	nil	variable	0	percent is-assigned-by equals to nCount is-cast-to float and nTotal is-cast-to float are-divided , percent is-returned	negative

Table 5: "variable" learning space for positive and negative examples

The similarity engine is then able to match artefacts between the positive and negative sets. To achieve this, weightings are applied to a subset of the learning space attributes. These weightings are based on domain knowledge. For example,

for variables, name, type, action and declaration-type are given the following weightings respectively: 4, 5, 1 and 4. Once the similarity engine has cross-matched variables and functions it is able to assert the differences into the "long-term" memory of the evaluation agent. Only the differences, i.e. the positive and negative pairing(s) that differed between examples, are recorded as classified example(s) of the given rule. Table 6 shows the similarity ratings for the artefacts described above. It shows that nCount is unaltered between examples; however both nTotal(total) and fPercent(percent) have been modified and therefore these pairings are then added to long term memory, as classified examples of the variable naming rule.

		Negative		
		nCount	total	percent
	nCount	100	45	14
Positive	nTotal		45	14
	fPercent			45

Table 6: Similarity of artefacts in variable learning space

5.4 Learning mechanism

A number of algorithms used to induce rules are now described and compared. To compare such algorithms, a metric is needed by which their performance can be measured.

To measure the performance of the algorithms over time, they are tested against training-data sets of different sizes. Each algorithm is run three times and the average of the percentage of positives covered by "correct" clauses is calculated.

The test data used to teach these algorithms is not structured in any way. Structuring the data would, in most cases, reduce the number of examples needed to induce the

correct rule, but it was important to maintain the unstructured nature of the data for these trials.

The learning algorithm used within the first detection agent is based upon the Generic Separate and Conquer Algorithm (GSCA) presented by Nils J. Nilsson [93, Nilsson]. This particular algorithm was selected because of the power and flexibility that accrues from its simplicity.

The GSCA algorithm is able to induce rules of the form:

$\alpha_1 \wedge \alpha_2 \wedge \ldots \alpha_n \supset$ positive

The GSCA algorithm was modified so that it could handle attributes that had continuous / discrete values (not just binary). It was also modified so that instead of producing multiple rules to cover all the positive examples, it produces a single rule that uses disjunctions to cover all the positive examples.

The modified GSCA algorithm (algorithm #1) is able to induce rules of the form:

$\gamma_1 \vee \gamma_2 \vee \ldots \gamma_n \supset$ positive

where γ is of the form:

$\alpha_1 \wedge \alpha_2 \wedge \ldots \alpha_n \supset$ positive

When applied to the learning data, the algorithm is able to detect defects simply by looking for learning data that are not covered by the rule.

5.4.1 Rule refinement

Due to the greedy nature of the learning algorithm, a refinement stage was included to remove any redundancy in the induced rule. This removed any attribute-value

pairings from a clause whose coverage was a subset of another pairings' coverage in that clause. This procedure also removed any clauses whose coverage was a subset of another clause's coverage.

5.4.2 Applying Rules (defect detection)

When directed by the user, the system is able to review a module of code. To facilitate defect detection, the detection agent receives from the collator agent the inputted source code described in all the learning spaces (currently variable and function). Then, from each learning plane where the agent has previously generalised a rule, it attempts to find contradictions in the data supplied. By implication the data first needs to be formatted according to the learning plane. Upon completion of defect detection for all rules, all contradictions (possible defects) are reported to the evaluation agent, which in turn displays the defects to the user via the source feedback window as shown in Figure 12. These defects are put forward with a confidence rating - the agent's own confidence in the rule that found the defect. There are a number of factors for determining system confidence in a defect, as discussed in section 4.4.10. The user is able to contradict or confirm a specific defect or confirm a review. This reinforcement / refinement by the user is necessary to ensure the knowledge and learning of the detection agents is "on track", and has a bearing on the confidence of the system.

5.4.3 Observations

The learning algorithm was tested against a number of training data sets, which represented a number of concepts. The concepts fell into three distinct categories:

those the algorithm was able to induce correctly within a reasonable amount of time, those it was unable to achieve; and those it was able to induce, but was unreliable or took a great deal of data to achieve.

The algorithm was reasonably adept at learning rules that comprised clauses containing one attribute-value pairing, for example *"init-before-use"* where at the function scope the rule induced was:

For all variables where (not...

(action[0]='inited') or (action[0]='assigned') or (declaration-type='parameter')

) = 'init before use' defect.

The data for this concept is considered distinct, as each element falls into one and only one clause.

The general trend was of increasing accuracy over time; however, the accuracy was not considered high enough for this system. It was apparent that the introduction of data that fell into "new" categories had a drastic effect on its performance. The 17th example included a parameter variable. The algorithm needed two further examples of a parameter variable to recover. Although it is reasonable that an algorithm needs a set of examples to be able to draw the necessary inferences, it is not acceptable that an algorithm be so distracted by the addition of one new datum.

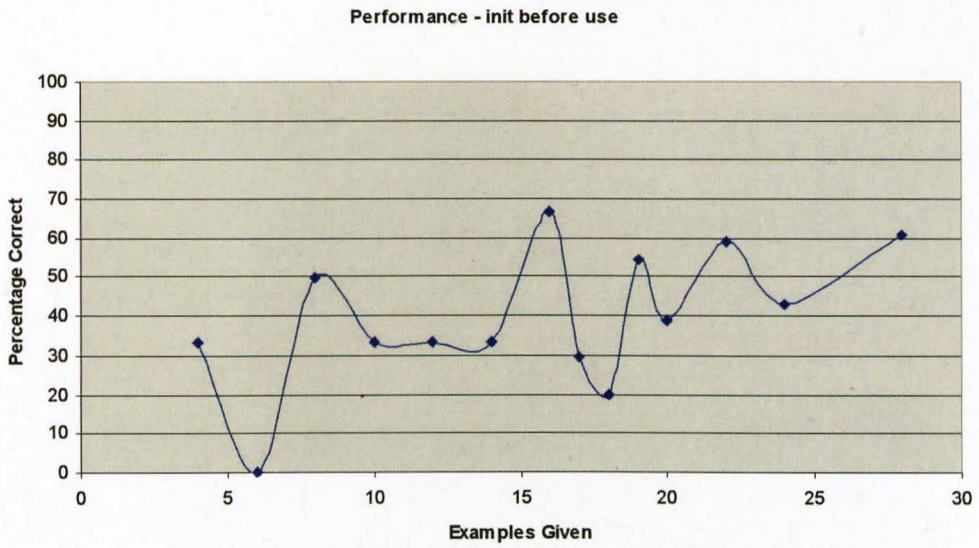


Figure 25: Performance for "init-before-use".

For rules such as *"unused variables"*, where the commonality is in the defects and not the positive examples, the induced rule was incorrect during the early stages where the number of examples was low. The rule finally induced was:

For all variables where (not

(action[0]='used') or (action[0]='assigned') or (action[0]='tested') or

(action[0]='inited')

) = 'unused variable' defect.

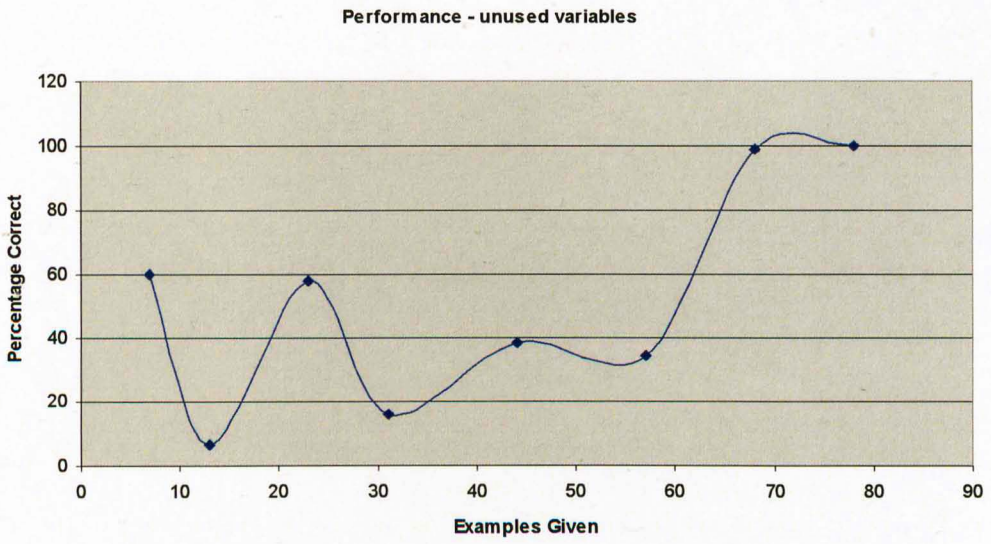


Figure 26: Performance for "unused-variables".

Over time, the induced rule was able to achieve 100% accuracy. This was because the system was able to easily determine if a variable was unused by examining each variable's action list and using negation (i.e. an unused variable is not a "used" variable). If a variable's action list did not start with inited, assigned, used or tested it must be unused. The system was able to do this because the learning plane abstracted actions into four categories of usage. However, a simpler pattern could have been identified in the negative example set, as all unused variables had an empty action list. It is feasible to assume that there will be other concepts whose commonality is in the defects. However, not all will have a clear complement in the positive set, in which case the algorithm would be unable to induce an appropriate rule.

For rules such as "variable naming" where the rule induced should have been of the form:

For all variables where (not

((name[0]='n') and (type='int')) or (name[0]='l') and (type='long') ...)

) = 'variable naming' defect.

It was unable to achieve a satisfactory conclusion.

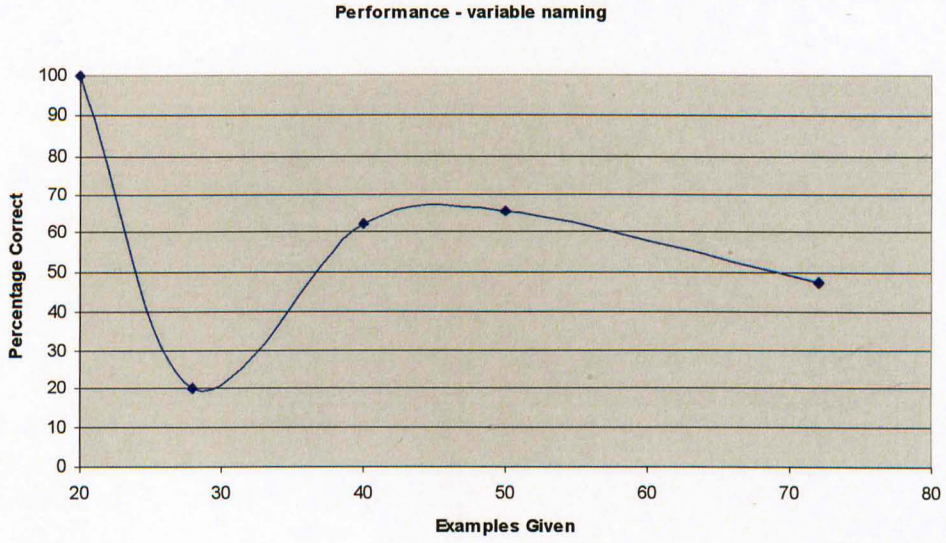


Figure 27: Performance for "variable-naming".

Over time, the accuracy of the induced rule fell as the noise in the data increased with the number of examples given.

The problem was identified as being a mixture of two facets of the learning algorithm when trying to learn this type of concept. The first was the method of calculating the "best attribute", where it used the net amount of positive examples that matched the attribute–value pairing (i.e. number of examples in the positive set that satisfy the pairing – number of examples in the negative set that satisfy the pairing). After time this resulted in the algorithm being swayed by noise. In this case the pairing name[4]='e' was selected as a best attribute after 28 examples were

given, as this was common across variable names regardless of type. The second was the way in which matches for each clause were removed from the learning set after each clause was completed. In this case, as the first clause was wrong due to the calculation of the best attribute selection, the remaining data was a misrepresentation, which meant that further attribute-value pairing selection was not optimal, and often detrimental. This situation was unrecoverable and unavoidable with this algorithm.

5.4.4 Improving accuracy

To address the concepts that algorithm #1 coped with but took a long time to arrive at the core of the rule, it was modified to have zero tolerance. Instead of allowing attributes that "mostly covered" the positive examples, it selected the attributes that "only covered" the positive examples.

This modified algorithm (algorithm #2) is able to induce rules of the form:

$\gamma 1 \vee \gamma 2 \vee \ldots \gamma n \supset$ positive

where γ is of the form:

$\alpha 1 \wedge \alpha 2 \wedge \ldots \alpha n \supset$ positive

where at most only one rule is produced.

It is possible and acceptable for this algorithm to fail to produce a rule. However, when a rule is produced by this algorithm it is generally more efficient and more generic than that provided by algorithm #1.

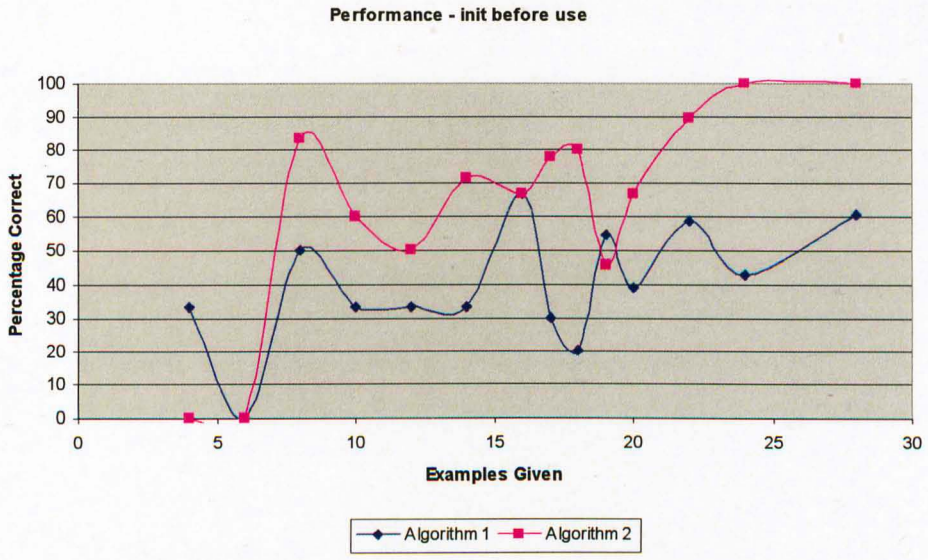


Figure 28: Comparison of algorithms 1 and 2 for "init-before-use".

As can be seen, the second learning algorithm out-performed the first quite significantly for concepts such as init-before-use (where the data is distinct), achieving 100% accuracy after 24 examples.

5.4.5 Improving coverage

To address the concepts that algorithm #1 struggled with, the algorithm was reversed so that, instead of looking at what the good code examples had in common, it looked at what the defects had in common.

Modifications to algorithm #1 meant that learning algorithm #3 is able to induce rules of the form:

$$\gamma_1 \vee \gamma_2 \quad \vee \ldots \gamma_n \supset \text{negative}$$

where γ is of the form:

$$\alpha_1 \wedge \alpha_2 \wedge \ldots \alpha_n \supset \text{negative}$$

When applied to the test data it is able to detect defects simply by looking for data that are covered by the rule.

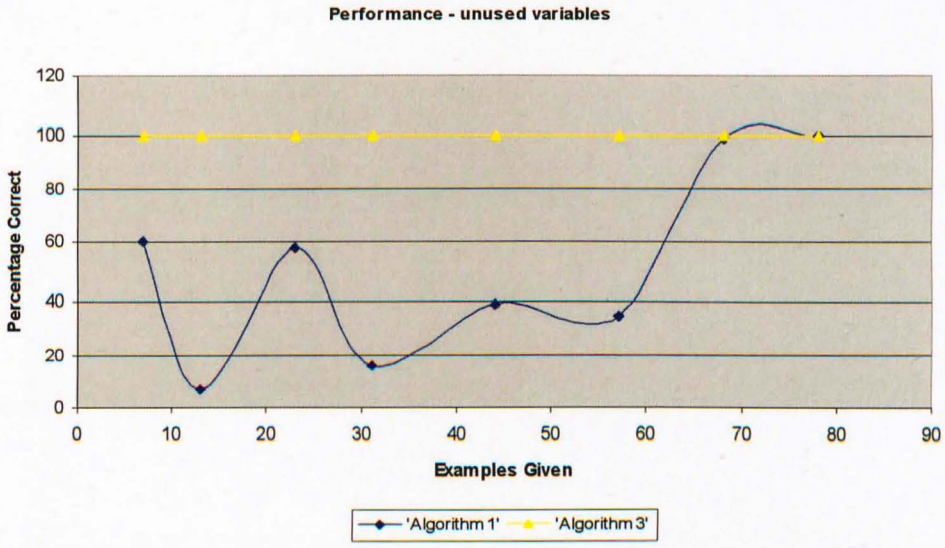


Figure 29: Comparison of algorithms 1 and 3 for "unused-variables".

For rules such as unused-variables, learning algorithm #3 performed well. In the specific case of unused-variables it was remarkably accurate (able to achieve 100% accuracy with only a few examples provided) due to the simplistic nature of the concept to be induced: an unused variable is one which has an empty action list.

5.4.6 Improving noise tolerance

To combat the detrimental effect on rule induction over time, caused where too much noise was introduced due to the number of examples, an improved method of calculating the "best attribute" was required.

For example, when given the same training data, algorithm #1 is unable to distinguish between name[0]=n and indirection=0 as the first attribute-value pairing to select, as both have a net performance of 16.

```
name0 n      P=38, p=17, N=32, n=1, diff=16.
indr  0      P=38, p=30, N=32, n=14, diff=16.
```

where P is the number of uncovered positive examples, N is the number of uncovered negative examples, p is the number of examples in P that meet attribute-value pairing and n is the number of examples in N that meet attribute-value pairing.

Using the following formula for calculating gain taken from Propositional FOIL [94, Quinlan] Gain Metric

```
Gain = p * (log2(p/(p+n)) - log2(P/(P+N)))
```

in the same situation, learning algorithm #5 correctly selects name[0]=n.

```
name0 n      P=38, p=17, N=32, n=1, diff=16, gain=13.581185.
indr  0      P=38, p=30, N=32, n=14, diff=16, gain=9.864432.
```

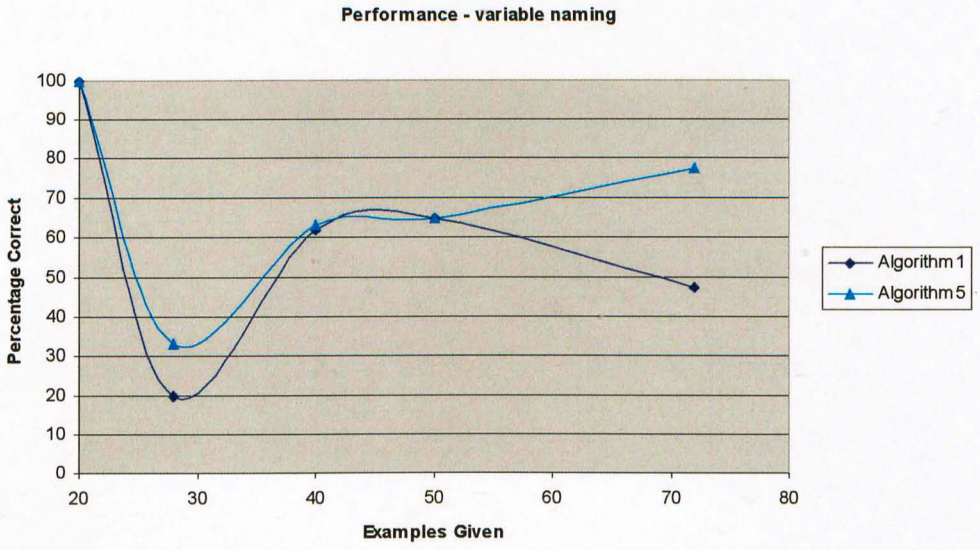


Figure 30: Comparison of algorithms 1 and 5 for "variable-naming".

As can be seen the performance of algorithm #5 is more robust over time, and it is able to tolerate noise far better than algorithm #1.

5.4.7 Observations

The algorithms were compared against each other for an example of each of the three rule types:

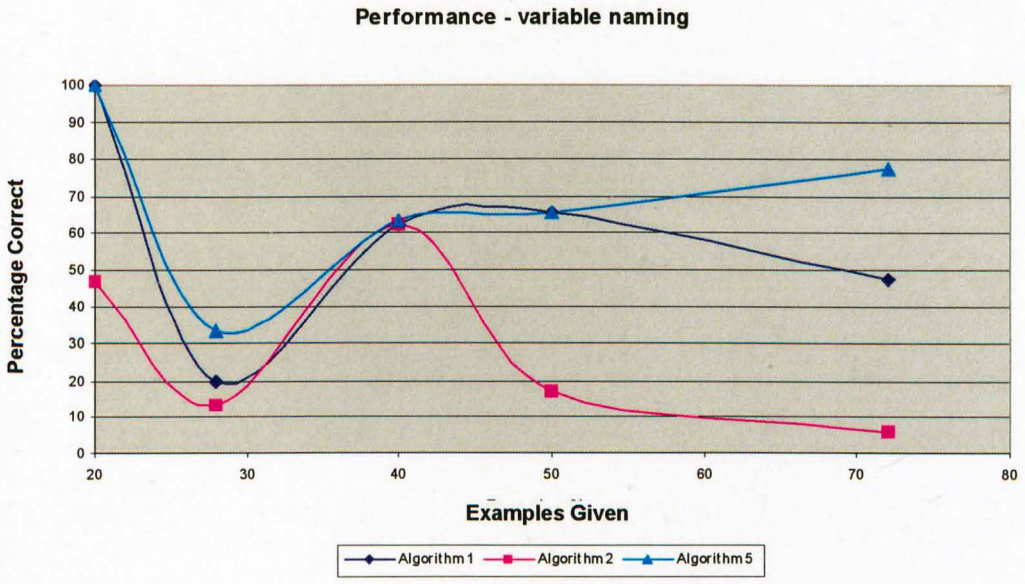


Figure 31: Comparison of algorithms 1, 2 and 5 for "variable-naming".

For rules that are susceptible to noise like variable naming, Algorithm #5 gave the best performance over time.

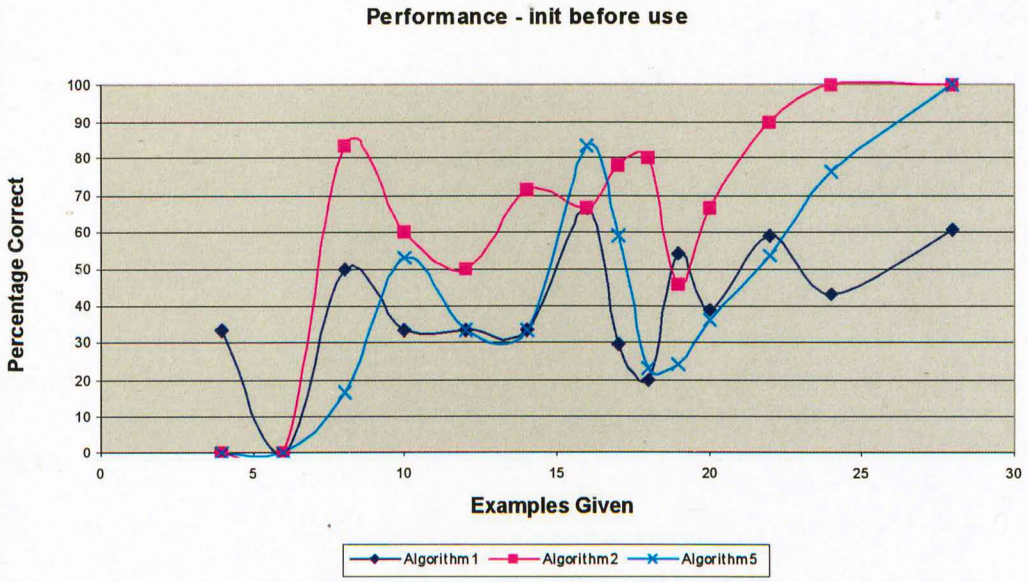


Figure 32: Comparison of algorithms 1, 2 and 5 for "init-before-use".

For rules that have distinct data, Algorithm #2 gave the best performance from the start. However, over time, Algorithm #5 appeared to be able to achieve the same accuracy.

For rules whose data has commonality in the negative examples and not the positive ones, Algorithm #3 gave the best performance (Figure 29).

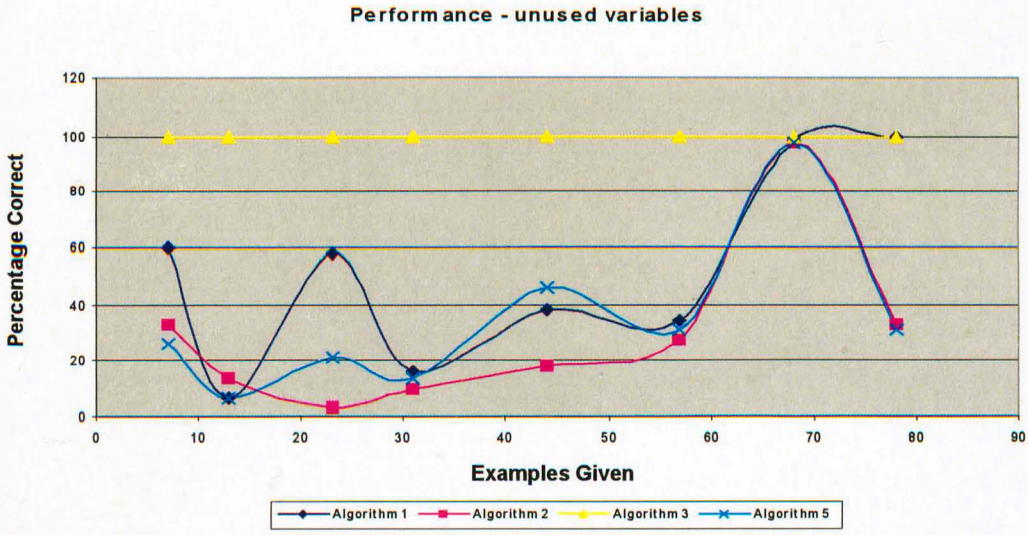


Figure 33: Comparison of all algorithms for "unused-variables".

5.5 Multiple detection agents

It was apparent that incorporating learning algorithms #2, #3 and #5 in the system would improve performance and usability. Instead of attempting to predict which algorithm would be best suited for the training data or combining them into one larger algorithm that exhaustively tried to find the best rule, it was decided that multiple detection agents could be deployed, each containing a different learning algorithm. This would allow the learning algorithms to stay separate (and therefore retain their simple / flexible nature) whilst providing the best coverage.

All the detection agents use the same long-term memory data (positive and negative examples collected over time), as obtained from the evaluation agent. This ensures that out-of-date information is not used and that the agents are learning from a consistent foundation. They also all attempt to learn the same rules. When given new examples of a guideline each agent is notified and relearns the concept from the new set. When directed by the user all agents review a piece of code, using their own induced rules to pin-point defects. Therefore in the current system with four agents and four planes, each guideline can have 16 possible generalizations describing it.

Results of testing the system that comprises detection agents implementing learning algorithms #1, #2, #3 and #5 are now presented. The input to the system is real life – project based code, which has not been seen before by the agents. However, it has been written by the same programmer as the training data. The full feedback is given to the system for each defect found – to see over time how the confidence mechanism adapts.

5.5.1 Observations

When given further examples of the three test concepts it was initially apparent that the noise tolerant algorithm implemented within DetAgent5 is alone sufficient to provide the learning mechanics of the system. It out performed the other algorithms not just on the type of concept it was developed for (variable-naming), but also for init-before-use.

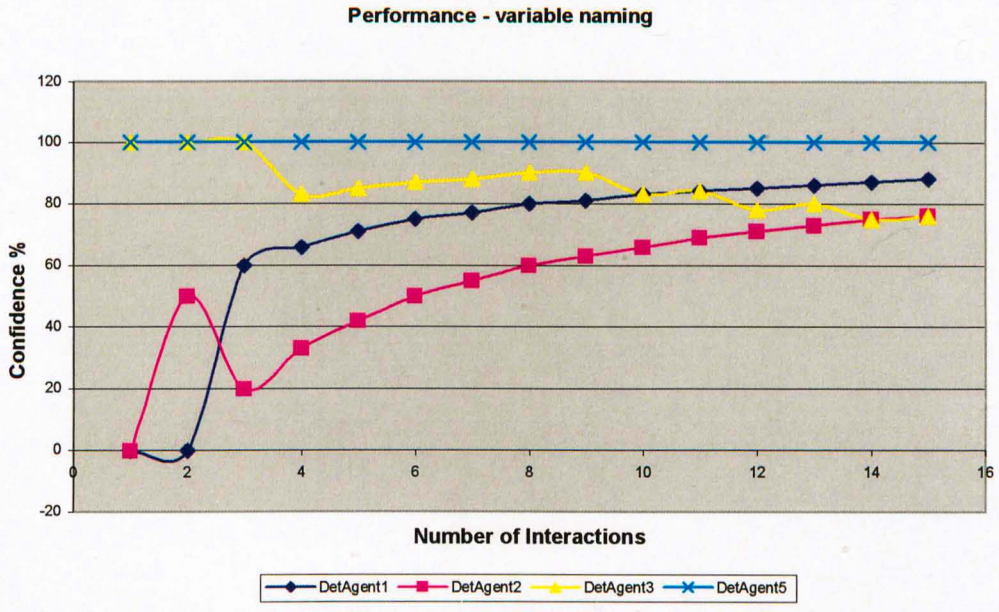


Figure 34: Performance with user feedback for "variable-naming".

When user feedback was taken into account for the variable-naming concept, the evaluation agent's confidence grew over time in all the agents except for DetAgent3 whose defects were often contradicted. DetAgent5 was never contradicted, and was able to remain 100% accurate for all of the examples given.

When learning "init-before-use" (Figure 35), DetAgent5 was again the best performer over time with the highest confidence rating after 10 examples.

DetAgent2, which implemented the algorithm developed for this type of concept, also performed well.

For the concept of unused-variables (Figure 36), DetAgent1 and DetAgent3 were able to maintain 100% accuracy, where DetaAgent2 and DetAgent5 initially faltered. Given the initial training set, it only took a few (3-5) interactions for each agent to induce the final rule.

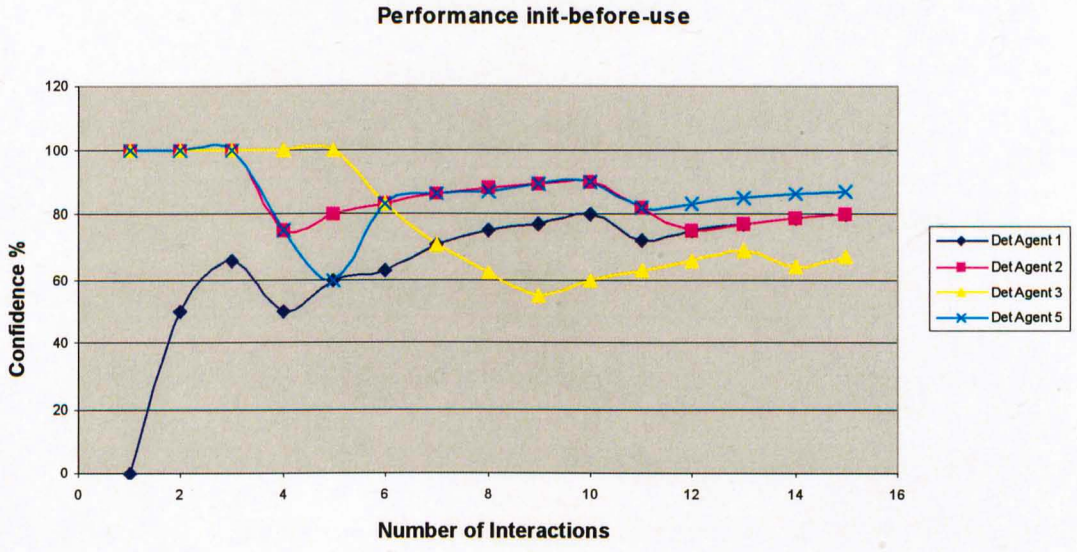


Figure 35: Performance with user feedback for "init-before-use".

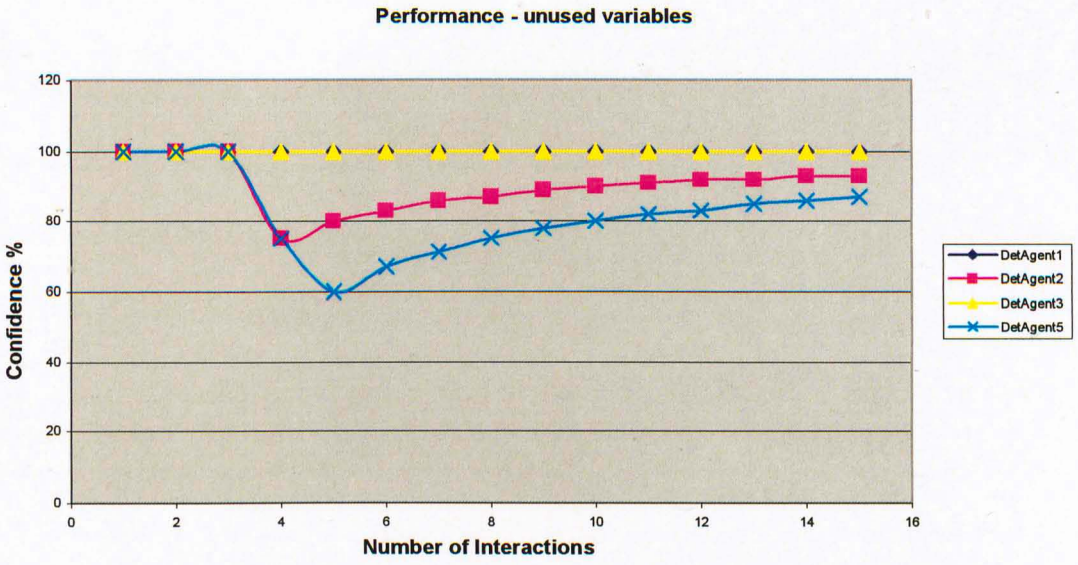


Figure 36: Performance with user feedback for "unused-variables".

Problems faced during testing included the adequacy of the test data and the interpretation of the systems results:

Inadequacy of test data - the source code was in pre-review state. This meant that certain rules, such as variable naming, had not been strictly adhered to, which gave the system a good set of positive and negative examples to work with. However, in the case of init-before-use, which is a concept of correctness rather than of style, there were fewer examples.

Interpretation of system results – there were a number of unexpected differences between the assumptions made before testing and what the tests highlighted. For example, the concept induced from the training data for 'unused variables' was not actually correct, as it classified variables as used if they were only initialised or assigned. In the context of real code this was more apparent as a defect. Also, the concept initially induced for init-before-use was incorrect as it triggered on unused variables.

5.6 Discussion

The results from testing the four algorithms indicate that, to allow the system a reasonable chance of inducing a rule that fits with relatively few examples, the direct algorithms are needed - ones that work on net coverage. Then, over time, algorithms that perform better against noise are needed to ensure that learning stays "on track".

The results from testing the algorithms with user feedback were encouraging as they showed a marked improvement in the accuracy of the system.

Chapter 6

Relational learning

The previous chapter described how the use of multiple learning algorithms improved the performance of the system. However, only relatively trivial bugs could be learnt with the algorithms described (i.e. they only improved the system's ability to learn a subset of the guidelines required if the system were to be deemed usable). The system would be unable to adapt to detecting more traditional bugs. In this next section we present an improved learning algorithm that incorporates relational information, with the aim of increasing the possible intricacy of the concepts induced, and therefore increasing the scope of the system by allow more complicated defects to be found.

To identify the inadequacy in the learning algorithms described in the last chapter, the following guideline was analysed to discover what the rule would be in the

context of the 'function' learning space: "When calling CreateFile, the return value (handle to the opened file) should be checked for success before it is used". This requires that if the function CreateFile is seen to be called, the return value is captured and, before it is used, it is checked against a failure value. Implicit in this requirement is the need to calculate relations between the pieces of knowledge within the function learning space.

In general, relational pattern recognition rarely uses machine learning techniques [98, Pearce], but research has been conducted to combine these approaches in the domain of recognition systems [95, Califf & Mooney][96, Craven & Slattery]. Adrian Pearce of the University of Melbourne has carried out a great deal of research in the relational learning field in conjunction with the Australian department of defence and others [97, Pearce et al][100, Pearce][99, Perugini et al].

In their work Pearce et al have shown that by taking into account the joint occurrence of unary and binary features, the search space can be constrained. For example in facial recognition, it is not only the existence of a nose, a mouth and a given distance between two unspecified parts that are important, but also the joint occurrence of the nose and mouth in a specific relationship (part I – relation IJ - part J) [98, Pearce]. This is referred to as part matching.

It is this mixture of unary and binary attributes that will allow us to induce rules that describe bugs such as that described in the previous section. For example, part (statement) I = "hFile is assigned the return value from CreateFile", part (statement) J = "hFile is tested against INVALID_HANDLE_VALUE" and relation IJ = the next thing to happen to hFile after I is J.

If we define a unary feature space U_I which represents the set of feature attributes from all statements in a function and from all patterns (defect-rules) $\{U_I=s_I : I=1,\ldots,n\}$, and a binary feature space B_{IJ} which represents the set of feature attributes from the relationship between pairs of statements within a function, from all patterns, $\{B_{IJ}=s_i s_j : I=1,\ldots,n ; J=1,\ldots p\}$, we can show, as in Figure 37, the generated unary rules u_1 and u_2 in feature space U_I, and the binary rules b_1 and b_2 in feature space B_{IJ}. The evidence rules weightings for u_1, u_2, b_1 and b_2 for pattern 1 are 0.83, 0.25, 0.20, 1.00, and for pattern 2 are 0.17, 0.75, 0.80, 0.00 respectively. As can be seen, the generation of rules is more effective in the binary feature space, as their weightings are more acute (b_1 for pattern 2 = 80% coverage, b_2 for pattern 1 = 100%).

The non-rectangular nature of the rules (feature states) in the unary feature space denotes the specific nature of the induced rules. As a general conclusion could not be drawn, the rules list individual features which are unique to the positive set.

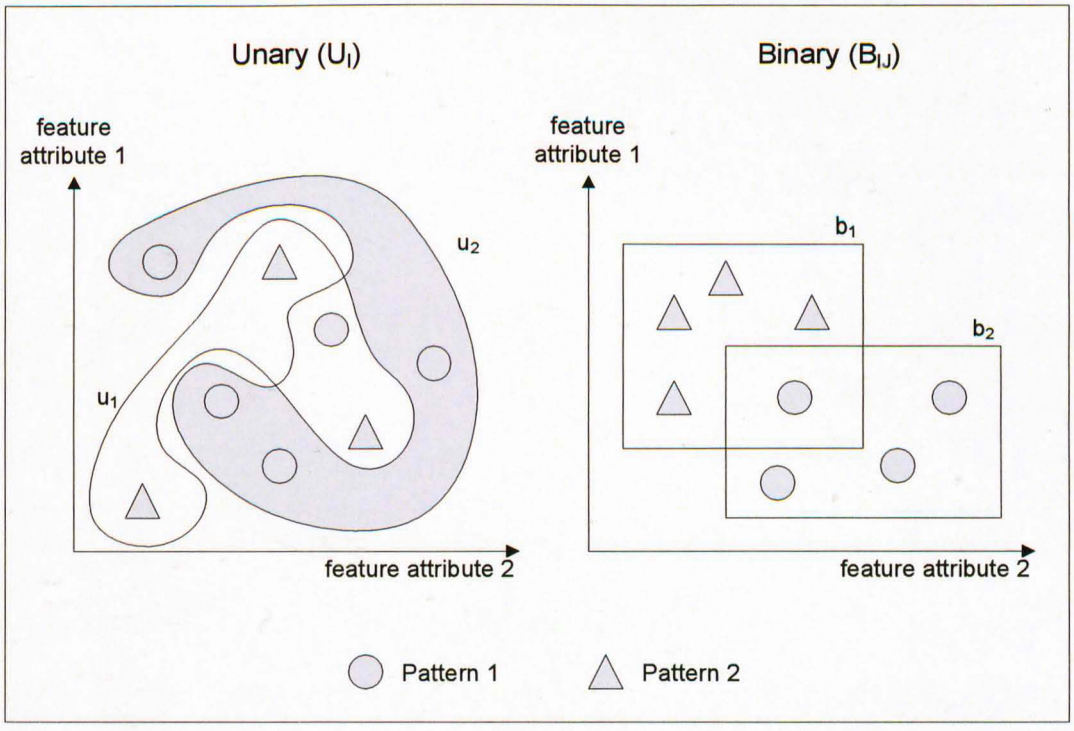


Figure 37: Comparison of unary and binary rule generation.

There are two common approaches to including relational information within a learning algorithm. The first is to extract the relational information as part of attribute selection and formatting, which allows presentation of the relational data to the learning algorithm. The second is to alter the learning algorithm to calculate and compare relational data from the original attributes supplied.

CLARET is a system for relational learning developed by Pearce. One application of the CLARET algorithm is the classification of hand-drawn symbols. The algorithm attempts to match an unknown example to one from a set of known examples using the follow stages: generate relations, attribute partitioning, part matching, and relational extension. The generate relations stage is where relational descriptions are generated by extracting relationships between pattern segments (of the hand drawn example). It is this aspect of the system that was considered for incorporation into the prototype system.

The CLARET algorithm only works on numeric values, as the attribute partitioning stage utilises numeric feature attributes of relations to form rules (feature states) by specialisation. In the programming domain the information supplied is not numeric but symbolic, as shown:

Name	Type	Decl	Indir.	Value	Action
hFile	HANDLE	variable	0	-1	identifier hFile is-initialised-to constant -1 is-declared , identifier hFile is-assigned-by equals to identifier CreateFile is-called-with identifier szFilename , identifier hFile is-not-equals-by constant -1 is-tested , identifier hFile is-used-as-argument

Table 7: Example of a LDO in the 'variable' learning space

This system deals with symbols (attributes) which are typically not numeric; therefore it is reasonable to assume that simply extracting relational information as

part of the attribute selection is insufficient. Relations would be missed for non-numeric attributes. For example, how could a numeric value be placed on the relation between two variable names?

An alternative to extracting relations during attribute selection would be to modify the learning algorithm. This would allow the system to process non-numeric attributes. However, without domain knowledge to guide the system, effective relations would be difficult to extract as the number of possible relations between two attribute values would add considerable unwanted noise to the data.

Therefore an approach is needed where information can be extracted during attribute selection such that an adapted learning algorithm can process this information as relations.

6.1 Extracting relational data

Domain knowledge dictates that by extracting relations between statements and variable-references (actions or sub-expressions made upon a single variable in a statement) using the control flow of the function, additional information about the behaviour of the code will be captured (i.e. above that which is described in the learning space).

Therefore, to include this relational data into the system, we extract relations from the learning planes and present this information as additional attributes. When preparing data for relational learning, the collator agent performs an additional step: relate attributes. This function allows the system to capture the relationships between different facts. The most obvious example is the control flow of the

function. As shown in Figure 38, a directed graph or tree can be generated to show the control flow of the function.

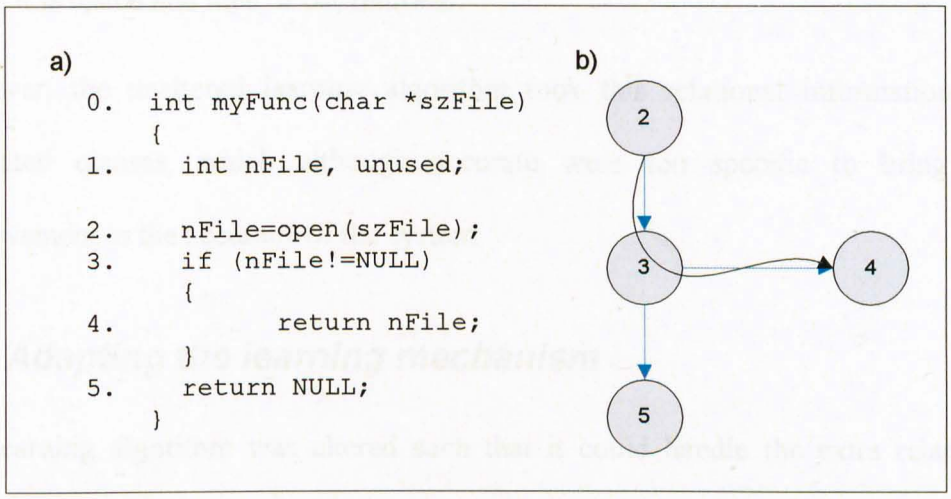


Figure 38: Function and graph of statements

By linking the nodes whose associated statement references a particular variable (a variable-reference), a path through the tree that represents the use of the variable can also be derived. This path shows the relations between the actions applied to / from the point of view of a single variable.

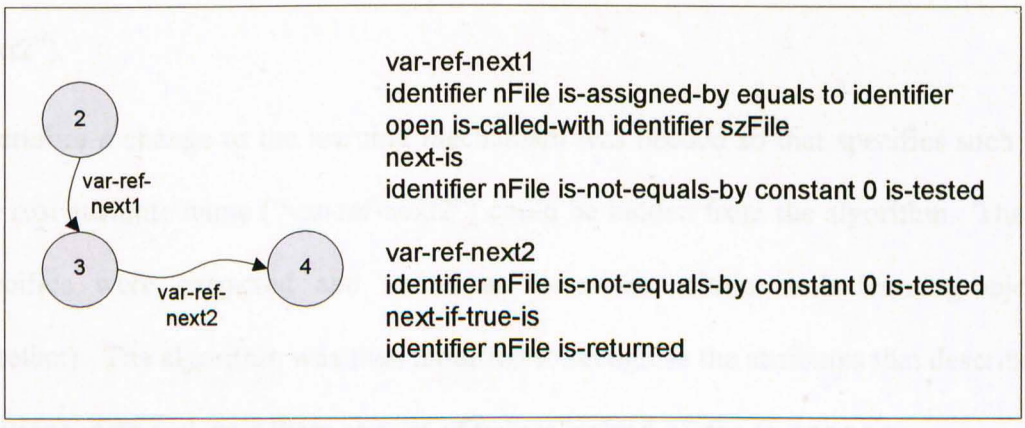


Figure 39: Variable reference graph

Figure 39 shows the two instances of the "next-is" relation for nFile. Firstly where it is assigned the return value from the open function and then tested, and secondly, where it is tested and then, if ok, returned.

However, the unaltered learning algorithm took this relational information and generated clauses, which although accurate were too specific to bring any improvement to the accuracy of the system.

6.2 Adapting the learning mechanism

The learning algorithm was altered such that it could handle the extra relational attributes supplied by the attribute-selection. The explicit link between the attribute's name and its value had to be weakened. For the types of learning described in the previous chapter the link between the variable name and its value was imperative, i.e. "name" = "hFile". However, this is too restrictive for relational learning. Many values may be supplied for each relational attribute (e.g. in Figure 39, there are two values for the relation "var-ref-next"; "var-ref-next1" and "var-ref-next2").

Therefore a change to the learning mechanism was needed so that specifics such as the real attribute name ("var-ref-next2") could be hidden from the algorithm. These specifics were extracted and stored as meta-data within each learning-object (artefact). The algorithm was then modified to recognise the attributes that described relational data and treat them as a set of values instead of one-to-one pairs.

6.3 Evaluation

The relational learning mechanism was implemented within detection agent 4. The "function" learning plane was altered to support relational learning. The system was then tested against a set of example code fragments. The fragments were not specially ordered or structured, but did cover, without duplication, most configurations of the code surrounding the concept to be learnt.

The concept chosen was the one described at the beginning of this chapter: "check-return-values". The code fragments were all concerned with checking the return value of CreateFile, the Windows equivalent to the open function. There are three main error handling strategies that can be employed: "nested-on-success", "check-and-bail", and "bail-and-tidy". Examples of these strategies, as applied to the CreateFile function, are given:

```
hFile=CreateFile(szFilename);
if (hFile != INVALID_HANDLE_VALUE)
{
  bRet = ReadFile(hFile, szBuffer, 32, &dwRead, NULL);
  // do other processing.
  CloseHandle(hFile);
}
```

Figure 40: "nested-on-success" error handling strategy

```
hFile=CreateFile(szFilename);
if (hFile == INVALID_HANDLE_VALUE)
{
  return FALSE;
}

bRet = ReadFile(hFile, szBuffer, 32, &dwRead, NULL);
// do other processing.
CloseHandle(hFile);
```

Figure 41: "check-and-bail" error handling strategy

```
do
{
  hFile=CreateFile(szFilename);
  if (hFile == INVALID_HANDLE_VALUE)
  {
    break;
  }

  bRet = ReadFile(hFile, szBuffer, 32, &dwRead, NULL);
  // do other processing.

} while (FALSE);

if (hFile!=INVALID_HANDLE_VALUE)
{
  CloseHandle(hFile);
}
```

Figure 42: "bail-and-tidy" error handling strategy

The system's performance was evaluated in two main areas. Firstly, the ability to induce a concept from the unstructured data, and secondly, the accuracy of the induced rules when used to detect defects in other examples.

Given that a concept or rule is constructed by clauses and that each clause in turn has sub-parts or axioms, the agent's confidence of an axiom (λ) and of a concept (φ) was calculated using the following formulas:

$$\lambda = \left(\frac{\delta}{\gamma}\right) \times \left(\frac{\alpha}{\beta}\right) \qquad \varphi = \min(\lambda) + \left(\left(\frac{\max(\lambda) - \min(\lambda)}{\gamma}\right) \times \max(\delta)\right)$$

Where α is the number of positive LDOs covered by the axiom, β is the number of pairs supplied, γ is the number of times the algorithm was run and δ is the number of times an axiom was induced.

The accuracy of the concept (ρ) was calculated using the following formula:

$$\rho = \left(\frac{\#(p \cup P) + \#(n \cup N)}{P + N}\right)$$

Where P is the set of positive examples (defect free), N is the set of negative examples (defects), p is the set of examples the system declared as ok, and n is the set of examples the system identified as defects.

6.4 Observations

The ability of the relational learning algorithm to induce a suitable rule was promising. Given a set of only four pairings, the learning mechanism was able to identify one axiom with 75% confidence (λ=0.75). The single axiom reflected the author's preference for error handling; handle the error straight away by tidying up and exiting. The concept as shown had an overall confidence rating of 59.4% (φ=0.59375):

> (var-ref-next = identifier hFile is-double-equals-by constant -1 is-tested next-if-false-is identifier hFile is-used-as-argument)
>
> or
>
> ((var-ref-next = identifier szString is-used-as-argument next-is identifier szString is-used-as-argument) and (var-ref-next=identifier strlen is-called-with identifier szString next-is identifier strlen is-called-with identifier szString))

A graph is presented to show the overall confidences of each agent in the concepts learnt. As previously discussed and as expected, agent 4 performed well. As can be seen from the chart agents 1 and 5 were also confident in the rules they had constructed. However, both were distracted by the consistent use of CloseHandle, a Windows function to close the file and free any associated memory. They both

generalised rules which contained the axiom: "statement12 = identifier CloseHandle is called with hFile", which is only coincidental and not relevant to the concept in question.

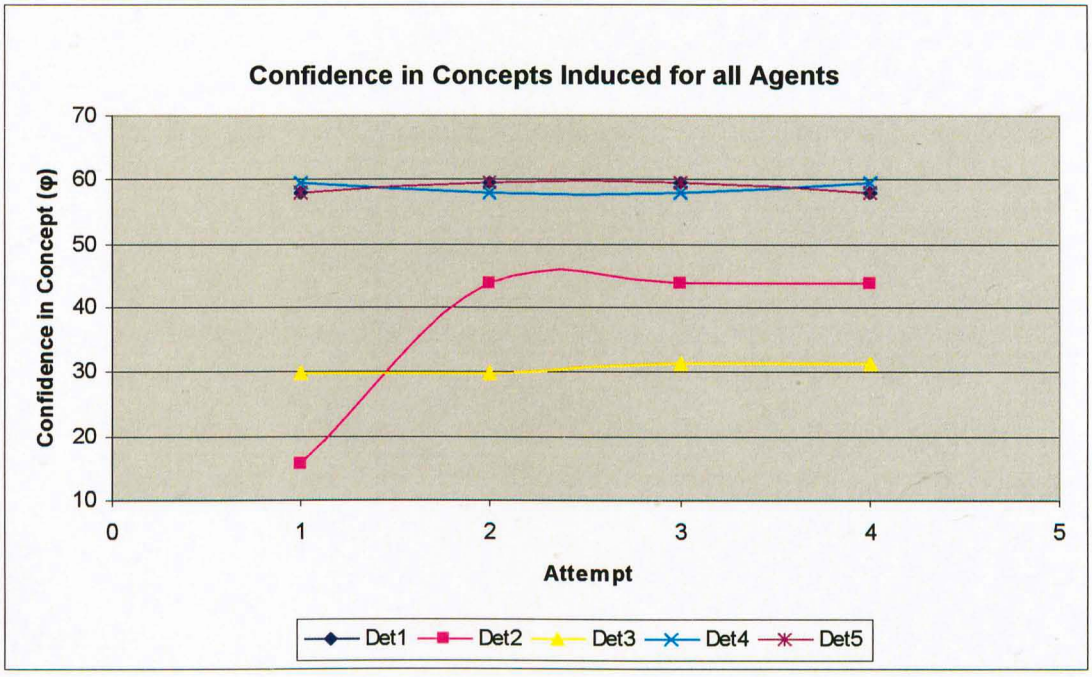


Figure 43: Confidence of concepts learnt for all agents

Detection agent 4 was used to review a number of other code fragments to see if the induced rule was effective. A further 8 fragments were supplied to the system. These fragments centred on a call to CreateFile. They varied in the error-handling strategy employed and general code structure (although, the variables used in the call to CreateFile, hFile and szFilename remained constant, due to the lack of a generalising function in the 'function' learning plane - see section 6.5). User feedback was not given to the agent as the rule needed to remain unchanged for the duration of this testing phase.

The accuracy of the rule on these 8 examples was 50% (ρ=0.5), as the agent incorrectly identified 4 examples, 2 of which were false positives.

Test	Description	Comment
1	A positive example where no defect should be found. This included an "act-on-error" error handling strategy.	Agent behaved correctly and no defect was found.
2	A positive example where no defect should be found. This included a "nested on success" approach to error handling.	Agent incorrectly identified the test as a defect, see Observation 2.
3	A negative example where there is no checking of the return value at all.	Agent behaved correctly, the defect was detected.
4	A bad example where a check is made, but it is against the wrong failure condition.	Agent behaved correctly, the defect was detected.
5	A bad example was given, where the return value was checked correctly, but the handle was used incorrectly.	Agent behaved incorrectly, no defect was reported, see Observation 1.
6	A good and bad example. This included two CreateFile calls, one was checked correctly, the other incorrectly.	Agent behaved incorrectly, no defect was reported, see Observation 3.
7	A good example where no defect should be found.	Agent behaved correctly and no defect was found.
8	A good example, This included a "nifty" trick of closing the handle as soon as the file is no longer needed, and not the more traditional way of closing the file at the end of the data processing section.	Agent incorrectly identified this as an error, see Observation 4.

Table 8: Results of testing learning algorithm 4

Observation 1: Incompleteness.

The inability of the agent to correctly identify the inappropriate use of the handle, after it has been correctly checked as shown is Figure 44, is in part due to the way the system uses the rules to identify defects. If the rule is satisfied by the data within the LDO no defect is identified. If the LDO does not satisfy the rule a defect is reported. In this example the rule is satisfied as the next-if-false statement after the check against INVALID_HANDLE_VALUE (-1) is the call to CloseHandle where hFile is used. However, the inappropriate use of hFile in the call to ReadFile is missed. This could either be fixed by creating another rule for appropriate use of variables, or by modifying the algorithm such that it captured implied information; if the call fails and hFile is equal to -1 then it should not be used again. This has

problems as the programmer may want to print out the value of hFile, which would be categorised as the variable being used and result in a defect being identified.

```
hFile=CreateFile(szFilename);       // Open the file..
if (hFile == INVALID_HANDLE_VALUE) // if NOT ok proceed!
{
  bRet = ReadFile(hFile, szBuffer, 32, &dwRead, NULL);
  // do other processing
}
CloseHandle(hFile);
```

Figure 44: Code snippet: incorrect error check.

The preferred solution would be to implement a relational learning agent that looks at the commonality in the negative examples similar to detection agent 3 (see section 5.4.5).

Observation 2: Inverse logic

The coverage of defects would be increased if the system were to use logic. For example, the induced rule dictates that if the variable is compared for equality to -1, and then if-false it is used, the inverse could be deduced using deMorgan's laws; if the variable is compared for non-equality to -1 then if-true it is used. This would identify the defects similar to the one missed in testing, the code of which is shown:

```
hFile=CreateFile(szFilename); // Open the file..
if (hFile != INVALID_HANDLE_VALUE)  //..if ok proceed
{
  dwRet = ReadFile(hFile, szBuffer, 32, &dwRead, NULL);
  // do other processing
  CloseHandle(hFile);
}
```

Figure 45: Code snippet: inverse logic check - "nested-on-success"

One possible solution to this is for the performance element to deduce the inverse version of a rule whilst looking for defects. Another is that the learning element

performs the extra processing and includes the deduced axiom as an extra clause to the concept.

Observation 3: Single good example hides defects.

It was observed that some defects were missed because of the presence of a single good-example. This is due to the method used to locate the defects as described previously. Once a good example has satisfied the rule no further processing is done to ensure other instances conform.

In the example shown in Figure 46, the good-example of checking the value of hFile against INVALID_HANDLE_VALUE and then if ok proceeding to use the value of the handle, absolves the whole function of the defect. The check at the end of the function although functionally correct, should have triggered as a defect, as this check is the inverse of the one the rule has been induced to recognise (see observation 2).

```
do
{
  hFile=CreateFile(szFilename);
  if (hFile==INVALID_HANDLE_VALUE)
  {
    break;
  }
  // do other processing.
} while (FALSE);

if (hFile!=INVALID_HANDLE_VALUE)
{
  CloseHandle(hFile);
}
```

Figure 46: Code Fragment "Bail & Tidy"

6.5 Discussion

The inclusion of a relational learning mechanism has significantly improved the usability and scope of this system. It is now able to learn concepts with the complexity necessary to address real programming issues.

It was apparent during this testing phase that some assumptions about dealing with learning in the 'variable' learning space were not appropriate for the 'function' learning space. LDOs in the 'variable' learning space either did or did not meet the rule, each LDO only described one variable and therefore further processing was not required. This is not the case for data in the 'function' learning space, where many instances may need to be checked in one LDO before we can ascertain whether or not a defect is present. The need to support constructs such as "for all" and "at least once" is needed. This would require a small modification to the learning algorithm. At the moment, for clauses using relational attributes, "at least once" is implied. This needs to be explicitly stated with alternatives made available.

The need for an agent capable of relational learning which looks at the commonality within the negative set of examples is apparent, in the same way that it was during the testing of the non-relational learning algorithms. This could be accomplished in a later prototype.

Once these issues have been resolved the 'function' learning plane can be extended and more learning planes created to give a wider scope, upon which concept learning can be facilitated, ensuring the range of bugs that can be detected will be acceptable to programmers.

Chapter 7

Performance & Evaluation

The prototype system has been tested at various points during the development. The results have been presented in sections 4.5, 5.5.1 and 6.3. The testing now focuses on the original hypothesis, which was:

"To determine if it is possible to capture the best practice from the peer code review process, such that a system can act as a consultant on best practices and learn the fundamentals of good programming practices, and to examine whether an agent based system could achieve this".

The main requirements for the system, as listed in section 2.4 were:

- The system should not be able to modify the code in any way. It should be offline. Advice should be available only upon request by the user.

- It is not adequate to simply collect metrics; assistance is needed in the form of detailed explanations / justifications of recommendations or the provision of good examples / solutions.
- Flexibility is needed between projects. Rules need to be flexible as some practices result in stylistically correct, maintainable code, but others may provide a more suitable solution.

In an attempt to quantify the system's performance against these top level criteria, a further 3 phases of testing were performed: individual, team and full-system.

7.1 Stereotype Assignment

All three phases of testing used the same user base of 17 programmers from HMGCC, each varying in skill and experience. Prior to the testing, each was assigned to a stereotype group to reflect their perceived abilities, as shown:

Initials	Initial Stereotype
PB	Experienced Win32
MF	Experienced Win32
PT	Experienced Win32
JW	Learner
CH	Learner
MB	Learner
SC	Embedded
AL	Embedded
HJ	Embedded

Initials	Initial Stereotype
RN	LINUX
KT	LINUX
PD	LINUX
AGi	Win32
AGo	Win32
JA	Win32
PA	Win32
CW	Win32

Table 9, Initial stereotype assignments for participants

7.2 Individual testing

The individual tests were focused to answer the following questions:

- Is the system usable?
- Does the system give enough information to be useful?
- Does the system adapt to the user?

To do this, tests were aimed at assessing the system's ability to learn rules which describe real bugs, and how successfully it uses these rules to identify bugs and inform the user of its recommendations.

7.2.1 Ability to learn from an individual

The ability of the learning mechanisms to induce adequate rules is very promising as shown in section 6.4. To fully assess the performance of the system, the results from the evaluation of the learning mechanism were expanded upon. The aim of this test was to see how far the system could be pushed towards inducing a fully complete and usable rule.

A new learning plane using attributes from the function learning space was developed which only used relational information. The system was then given some preliminary data from which to induce a rule. The rule to be induced was the same as that discussed at the beginning of Chapter 6.

A second measure of the accuracy of the concept (additional to that given in section 6.3) was used (ρ^1). This value attempted to include the intolerance of the engineers to false-positives.

$$\rho^1 = \left(\frac{\#(p \cup P) + \#(n \cup N) - \left(\#(n \cup P)/2\right)}{P + N} \right)$$

Where P is the set of positive examples, N is the set of negative examples (defects), p is the set of examples the system declared as ok, and n is the set of examples the system identified as defects.

Given a limited set of four pairings, the learning mechanism was able to identify a concept composed of a single axiom with an overall confidence rating of 100% (φ=1), as shown:

> (var-ref-next=identifier hFile is-assigned-by equals to identifier CreateFile is-called-with identifier szFilename next-is identifier hFile is-double-equals-by constant -1 is-tested)

This rule partially describes the condition necessary for the error handling strategies: check-and-bail and bail-and-tidy.

When tested against a set of examples (14 positive and 5 negative) that the system had not encountered previously, it detected defects within the code with an accuracy rating of 52.6% (ρ=0.579, ρ^1= 0.526). It identified 12 positives, of which 9 were in the positive set and 7 negatives of which 2 were in the negative set, leaving 8 examples incorrectly categorised.

The induced rule did not include the action that should be taken, which is dependant on the result of the conditional statement. This is due to the learning set given to the system. The negative examples contained no error handling strategy. A further set of examples were supplied to the system which included defective or incorrect

examples of the check-and-bail and bail-and-tidy error handling strategies and the associated fixed versions. From them the following rule was generated:

(var-ref-next=identifier hFile is-assigned-by equals to identifier CreateFile is-called-with identifier szFilename next-is identifier hFile is-double-equals-by constant -1 is-tested **and** var-ref-next=identifier hFile is-double-equals-by constant -1 is-tested next-if-false-is identifier hFile is-used-as-argument)

or

(var-ref-next=the-address-of identifier dwWritten is-used-as-argument next-is identifier dwWritten is-not-equals-by identifier strlen is-called-with identifier szString plus constant 1)

This first clause of the rule describes what should happen after the return value is checked. The second clause covers the other single positive example, not covered by the first. The agent's confidence in this rule was 72.6% (φ=0.726). The accuracy of this rule was (ρ=0.737, ρ^1=0.605) 60.5%, this was calculated by its performance on the same test data as previously used. It identified 9 positives all of which were in the positive set. It identified 10 negatives, 5 of which were in the negative set. Of the five positive examples it identified as defects, 2 used the alternative error handling strategy, which had not been identified by the learning algorithm as it was only used in 2 of the 19 examples, and the other 3 examples used code syntax that the system was unable to understand. This is discussed in the next section.

Conclusions

The ability of the system to induce a rule as presented above, and be confident in it, was shown to be very good indeed. Not only is the system able to induce rules

which are complex enough to capture the information required, but the system is able to do so with an accurate measure of its own abilities. This equates to a powerful technique applicable in the real world, as the system is able to work effectively with small data sets, by having a measure of confidence in its findings. It is therefore less likely to mislead with erroneous recommendations.

7.2.2 User & System Interaction

The ability of the system to use the induced rules to identify defects, and the suitability of the information supplied to the user about such defects, is now evaluated.

When a defect is identified in the source code, the system denotes this in the feedback window, by highlighting the element to which the defect is related, as shown.

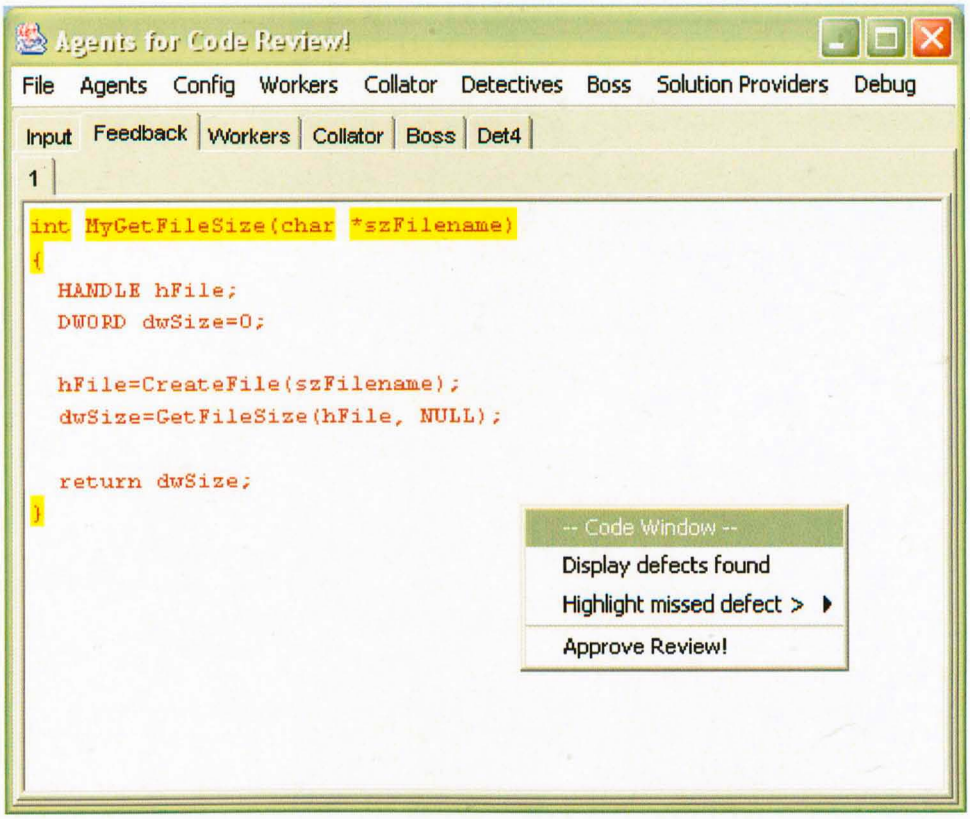


Figure 47: Screen capture of prototype system: highlighting defect

If the user clicks on this window a number of options are presented. The user is able to display details of the defect, highlight to the system any missed defects, or approve a review.

If the user feels the agents have spotted all defects in the code, the "Approve Review" option allows the agent to be 'rewarded'. This increases the evaluator (boss) agent's confidence in the agent(s) who recommended the approved defects. The user is able to 'punish' the agents, by highlighting a code element which should have had a defect reported on it. By selecting a guideline that the agents missed, the evaluator agent's confidence in all agents for that guideline is decreased.

If the user selects "Display defects found", the details of the defects that have been reported are displayed in the bottom of the window, as seen in Figure 47. There may

be more than one defect associated with the element highlighted. The defects are then listed in the output window, as shown.

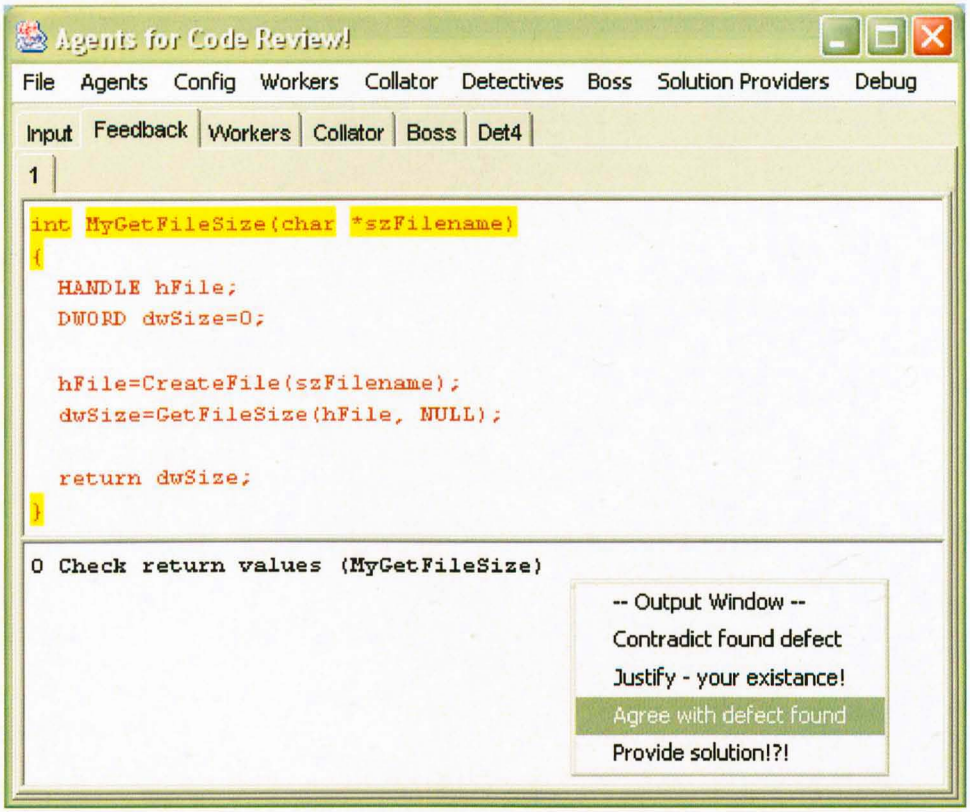


Figure 48: Screen Capture of prototype system showing details of defect

Once the individual defects are displayed in detail, the user is able to select "contradict found defect", "justify found-defect", "agree" or "provide solution".

If the user chooses to agree with a defect, the evaluator agent's confidence in the defect's origination agent(s) is increased. However, if the user chooses to contradict a found defect, not only is the evaluator agent's confidence in the defect's originators decreased, but the agents that reported the defect are instructed to relearn the concept from the evaluators long-term-memory, which now includes the current piece of code as a positive example.

If "Justify" is selected, the system attempts to explain its recommendation. This is achieved by displaying details of the supplied code and the rule's clauses, in the same learning space, as shown:

```
Det4Agent:justifying Rule = Check return values in
function(name=MyGetFileSize)

var-ref-next0=identifier hFile is-assigned-by equals to
identifier CreateFile is-called-with identifier
szFilename next-is identifier hFile is-used-as-argument
var-ref-next1=identifier dwSize is-assigned-by equals to
identifier GetFileSize is-called-with identifier hFile
and constant 0 next-is identifier dwSize is-returned

clause: var-ref-next=identifier hFile is-assigned-by
equals to identifier CreateFile is-called-with
identifier szFilename next-is identifier hFile is-
double-equals-by constant -1 is-tested and var-ref-
next=identifier hFile is-double-equals-by constant -1
is-tested next-if-false-is identifier hFile is-used-as-
argument
    => does not match.
clause: var-ref-next=the-address-of identifier dwWritten
is-used-as-argument next-is identifier dwWritten is-not-
equals-by identifier strlen is-called-with identifier
szString plus constant 1
    => does not match.
```

Figure 49: Trace output from prototype system showing justification of defect

As the rules have been generated from the user's history, the information that the system is trying to convey to the user should already be understood and known by the user. However, the manner in which the prototype currently displays the justification is not intuitive.

If the user selects "suggest solution" the system is able to extract the nearest positive example from the user's history.

```
Sol1Agent:providing solution for Rule = variable naming
(name=count)
Original = int count=0;
Solution = int nTotal=0;

Why? Clause = name[0]='n' and type='int'
```

Figure 50: Trace output from prototype system providing solution

In the case of variable naming, this allows the system to announce that: "int count' should start with an 'n', as shown in Figure 50. Although it can not say directly that it should start with a 'n', it is able to show an example nTotal, and then show the clause in the variable naming rule that states that any variable of type int should start with an 'n'.

Again, as this information has been learnt from the user, it is not trying to present something new to them, and as such a detailed justification is not required. This technique is less effective in providing solutions for defects in the function learning space, as each element in the function learning space can be constructed from many parts to which defects can relate.

Conclusions

The user interface for the prototype system is fit for purpose, with the raw information provided being sufficient for a user familiar with the system to be able to use it effectively. Further work could ensure that the information is presented in a more intuitive and aesthetically pleasing way.

Further work is needed in the way the system displays solutions to the user. As shown, it simply displays a good example (the nearest it can match to the problem from the positive set) and the clause that triggered the rule. This again will be intuitive if the rule has been learnt from the current user's data set, but not if the solution has been provided by another user's node.

The user feedback or evaluation mechanism works well, by approving or rejecting defects the mechanism quickly adopts the new information, and in some cases induces a new rule. What was also helpful to the user was the ability to set the

confidence threshold by which the evaluation agent reported the defects. This allowed the user to tweak the system to optimal effectiveness.

7.3 Team testing

The team tests were carried out to answer the following question: Can the system adapt to different users? A number of software engineers from HMGCC, all varying in programming skill and experience, were selected and asked to supply fragments of code. From these fragments, the system attempted to learn or induce concepts and to describe the same bug for each user. The induced concepts where then tested for accuracy across the whole set of examples supplied.

16 engineers were asked to fix pieces of code, as though they had been tasked with actioning the recommendations from a peer code review. This scenario allowed the engineers to apply their own style and preferences to the piece of code, whilst maintaining a consistent theme. Seven fragments of code containing defects, and the descriptions of the defects, were issued to the participants (numbered Test 1 to 7).

The aim of the testing was to see how well the system could cope with learning one rule from different people, having to cope with different styles, preferences and differing levels of ability and consistency.

It was assumed that the system would be able to learn from the experienced Windows programmers as they have had more exposure to code reviews and were therefore more likely to be consistent in the way they approach and present their code.

To ascertain whether or not the system can adapt to different users its performance had to be asssessed in two areas: the ability to understanding the source code, and the ability to learn from examples.

7.3.1 Ability to understand source code

The fixed code fragments were processed to test the system's ability to understand the source code. The results can be seen in Table 10.

Initials	Code Processed?	Comment
PB	Y	Hash defines had to be simplified, as the prototype only does one pre-processor parse.
MF	N	Unable to understand code.
PT	Y	All code processed and understood..
JW	Y	All code processed and understood.
CH	Y	All code processed and understood.
MB	Y	All code processed and understood.
SC	Y	All code processed and understood.
AL	Y	All code processed and understood.
HJ	Y	All code processed and understood.
RN	N	Unable to understand code.
KT	Y	All code processed and understood.
AGi	Y	All code processed and understood.
AGo	Y	All code processed and understood.
JA	Y	All code processed and understood.
PA	Y	All code processed and understood.
CW	N/A	The code fragments were not returned due to illness.

Table 10: Understanding the source code test results

Two engineers supplied code that the system was unable to understand. MF and RN both prefer to structure their conditional statements so that the function call whose return value is being checked is assigned within the if statement, as shown:

```
if ((hFile=CreateFile(szFilename)) ==
                                    INVALID_HANDLE_VALUE)
{
  break;
}
```

Figure 51: MF's preferred style of checking return values

The mechanism for recognising sub-expressions and establishing variable references is currently unable to support side-effects in conditional statements. Therefore, the system is unable to understand construct such as this, so the code fragments supplied by MF and RN are unusable. Other than some syntax corrections all of the other engineers' code fragments were correctly parsed by the system.

Conclusion

The system parsed the code well, but the translation between tree representation and knowledge needs to be completed as there were inconsistencies. A more comprehensive translation mechanism is needed to accurately describe the contents of the construct in pseudo code.

7.3.2 Ability to learn from different individuals

The second phase was to ascertain whether the system could learn from the examples supplied. One engineer was selected from each stereotype: PB, MB, AL, KT and JA. The ability of the system to learn from these engineers was then analysed in more detail.

Initials	Initial Stereotype	Description	Confidence in concept (φ)	Observation	Accuracy (ρ)
PB	Experienced Win32	It was assumed prior to testing that the system would learn from PB easily, as he has been involved in the code review procedure for 5+ years. He has a systematic, consistent and mathematical approach to coding.	85.7%	PB had employed a lateral approach in one solution. The system was unable to adapt to this single instance.	47.4%
MB	Learner	MB is a new starter; he has limited experience of coding outside of university and of code review. MB is keen to improve his coding style and learn more about good programming practices	100%	MB's simplistic approach gave a consistent structure, which was beneficial to the learning mechanism.	57.9%
AL	Embedded	AL is new to code review, having only experienced 2 previously, but is keen to improve his coding style and learn more about good programming practices.	16.5%	AL's style evolved during the testing, this is reflected by the low confidence rating of the system. Over the 7 tests AL provided 3 different error handling strategies	57.9%
KT	LINUX	KT is an experience programmer with a great deal of LINUX programming experience. He is well versed in the process of code review.	100%	KT's systematic approach allowed the system to learn a concept with 100% confidence.	57.9%
JA	Win32	JA has been with HMGCC for nearly two years, he has had a reasonable exposure to code reviews during that time. He is considered a competent junior engineer.	100%	JA's fragments were fully consistent in their structure.	57.9%

Table 11: Team test results for confidence and accuracy

```
PB's = (var-ref-next=identifier hFile is-not-equals-by
constant -1 is-tested next-if-true-is identifier hFile
is-used-as-argument) or (var-ref-next=identifier hFile
is-used-as-argument next-is identifier hFile is-
assigned-by equals to constant -1)

MB's, KT's and JA's = (var-ref-next=identifier hFile is-
assigned-by equals to identifier CreateFile is-called-
with identifier szFilename next-is identifier hFile is-
double-equals-by constant -1 is-tested)

AL's = (var-ref-next=identifier hFile is-double-equals-
by constant -1 is-tested next-if-true-is identifier
hFile is-used-as-argument) or (var-ref-next=identifier
hFile is-not-equals-by constant -1 next-if-false-is
identifier hFile are-passed-into identifier CloseHandle)
or (var-ref-next=identifier percent is-assigned-by
equals to identifier fraction and constant 100 are-
multiplied next-is identifier percent is-cast-to DWORD
is-assigned-to identifier towrite) or (var-ref-
next=identifier dwSize is-assigned-by equals to
identifier GetFileSize is-called-with identifier hFile
and constant 0 next-is identifier dwSize is-double-
equals-by constant -1)
```

Figure 52: Induced rules for team testing.

The induced concepts for MB, AL, JA and KT all identified check-and-bail and bail-and-tidy error handling strategies; indeed, the rules induced from MB, JA and KT were identical. The induced concept for PB identified the use of the nested-on-success strategy. This accounts for the different coverage of the rules.

Contrary to the prediction, PB employed three different methods of checking the return value. The first, used in the first four examples, was the nested-on-success approach. Once the complexity of the function increased a check-and-bail approach was adopted. A lateral approach to the problem was given in one example, where the call to CreateFile was outside of the do-while construct.

Conclusions

The learning algorithm did not prefer certain people over others; it was able to work with everyone. It was apparent that the aspects of the code that the learning

algorithm was examining to determine consistency were at a much lower level than those used by participants in a peer code review. It appeared that the majority of participants of a peer code review used more superficial measures of consistency (such as layout and commenting) to gauge overall consistency, where as the agents used the relations between constructs. For example, PB's code was considered by the team to be the most consistent. However, in one of the early examples, PB used a shortcut to close the handle of the file when it was finished with. This, although correct and probably the way the compiler optimises the instructions when building, was not consistent with PB's other examples - a factor the agents were more attuned to identifying than their human counterparts.

During the testing phase there was a noticeable improvement in coding style for AL and MB. Both the programmers enjoyed the testing and learnt from feedback given, refining their style and error handing approaches. This demonstrates the ability of the system to improve knowledge in a manner similar to that shown by the process of peer code review.

7.4 System testing (Simulation)

The full-system simulations were carried out to answer the following questions:

- Is the hypothesis capable of recognising trends in the team of users?
- Is the hypothesis capable of forming a consensus of guidelines?

Simulations were used as significant effort would have been needed to ensure the prototype was robust and user friendly enough for testing. These simulations attempted to quantify the amount of social information the system could capture, by

examining how the stereotypes and roles of the individual programmers within the team evolved, based purely on the selection of solutions.

A fragment of defective code and a set of solutions were supplied to the testers (Test 2), as if the system had presented them. The testers were then asked to select one of the solutions, and explain their decision. This was then repeated with a different test case (Test 5).

Initials	Stereotype	Preferred Solution	Explanation
PB(5)	Experienced Win32.	5	Didn't like the superfluous do-while-false loop and extra if the other options had.
MF(7)	Experienced Win32.	5	Was correct and succinct but not very expandable.
PT(6)	Experienced Win32.	5	For debug and compactness.
MB	Learner	1	For the comments.
AGi(4)	Win32	1	Wanted the extra parameter check that 1 gave.
AGo(3)	Win32	5	No magic numbers and for the extra parameter check.
JA(2)	Win32	1	Well commented, good use of debug and no magic numbers.
PA(1)	Win32	1	For the comments and casting convention.
RN	LINUX	4	For the correctness of the return value from 'strtoul'.
PD	LINUX	5	As it is more efficient and does not use redundant constructs like the do-while-false loop.

Table 12: Simulation Results for Test case 1 (based on Test 2)

Initials	Stereotype	Selection	Explanation
PB(5)	Experienced Win32.	1	Liked the parameter check in this example. Didn't like the redundant resetting of a stack variable in others.
MF(7)	Experienced Win32.	6	For clarity and correctness.
PT(6)	Experienced Win32.	5	Good commenting which gives a good level of confidence.
MB	Learner	5	Not over done, well commented.
AGi(4)	Win32	1	For the parameter check.
AGo(3)	Win32	6	For the layout and initialisation of the stack variables.
JA(2)	Win32	1	Well commented, good use of debug and no magic numbers.
PA(1)	Win32	1	For the comments and casting convention.
RN	LINUX	6	For clarity (lack of debug statements and brief comments).
PD	LINUX	2	For the clarity of the layout and conditions.

Table 13: Simulation Results for Test case 2 (based on Test 5)

With just two repetitions the system was able to identify a number of trends in the test results.

- For the first test case, the majority of the Win32 programmers preferred PA's solution over the others, as did they in the second test case.
- The experienced programmers in the first test case, all preferred PB's solution. However this was not reflected by their choices in the second test case.
- The Linux programmers preferred the more succinct examples for each test case. There was no direct correlation between the authors of the solutions selected in the two test cases.

- The learner preferred the solutions where the code was well commented.

From these observations and by analysing the commonality in the solutions provided it can be hypothesised that:

- Linux programmers look more for correctness than consistency, as a gauge of quality, whereas Win32 programmers look for consistency and adherence to rules.
- The preferred quality-indicators change as the complexity of the code fragment increases. The more the piece of code was required to do, the less succinct but more generic the error handling strategy became (as shown by the solutions supplied by PB).
- The overall consensus (concept that fits all solutions provided) echoed the change in style in response to complexity. For the first test case (when stereotype weightings are employed) the consensus leaned towards the nested-on-success strategy. When the complexity was increased, as in the second test case, the consensus adopted was the check-and-bail error handling strategy.

These results show that the inclusion of stereotypes into the system are beneficial, and can be used in the system's user model to aid the selection of solutions for the user. This in turn improves the effectiveness of the system.

7.4.1 Observations of the team and results

Whilst gathering the results it was apparent that not many of the programmers used correctness as a criteria for selection. More than half looked at the error handling strategy used rather than the actual calls and checks made.

It was apparent that:

- The Win32 programmers mostly preferred examples supplied by PA. PA's style of coding is fastidious; taking the coding guidelines to extremes, ensuring all numbers are defined, that all functions are commented, all parameters are checked etc.
- The experienced Win32 programmers where more pragmatic about their selections, they were not looking for consistency but for suitability of the code. In the first test example 5 was radically different from the others, as it closed the handle to file after the read, as it was no longer needed. This was considered by the Win32 programmers as risky from a future maintenance point of view as it was not a consistent way of dealing with the situation. However, the experienced Win32 programmers all selected this option above the others, as it was correct and efficient, noting that the compiler would optimise the code down to something similar.
- The LINUX programmers looked beyond the stylistic issues, much like the Experienced Win32 programmers. However unlike them, they didn't look for efficiency more than correctness and accuracy. For example, checking the return value from strtoul against 0 for failure is incorrect behaviour as 0 is a possible return value.

These results show that it is possible for the system to capture the team's preferences and give greater weighting to more influential members. It is worthy of note that the simulations identified social behaviour which was not recognised by the team.

Within the lab most individuals turn to PT for assistance with coding. He is the most experienced non-management member of the team, and is therefore readily available to assist junior members. It was therefore assumed his preferences would be seen in the programming and solution selections of the others. This was not however the case as most programmers preferred PA's coding. It has been theorised that although the programmers use PT for help and advice, they don't wish to include his code in one of their own projects. This may be because PT's code is more pragmatic and less consistent than others.

It appears the team have a higher standard for other people's code than their own. A possible explanation for this could be that initially they do not fully understand the piece of code, and therefore look to the adherence of rules to ascertain its suitability. Or possibly that constraints such as time and pressure can excuse lapses of consistency in their own code, but not the solutions provided by others (as they are considered static).

Another possible explanation for the tendency towards consistency rather than correctness may be that the team was too focused on the code review process, and not necessarily on detecting bugs. The awareness of code review and good style issues should not outweigh the importance of catching bugs.

A further observation is that the LINUX programmers were less likely to be swayed by style and individual preferences than their Windows counterparts. This may be

mainly due to the open source community within which LINUX resides and the frequency with which LINUX programmers are exposed to other's code, either to fix, build or modify.

7.5 Discussion

A review of the performance of each agent is now presented, with future work and improvements also discussed.

7.5.1 Architecture

It is common throughout the agent community for agents to be employed as a team (i.e. in multi-agent systems). Commonly, larger agents are employed on dispersed systems to interact with each other to attain their goals. It is less common for multi-agent systems to be used within one node.

By giving the agents the ability to request information from each other and reply to these requests, knowledge and data can be exchanged in an unstructured manner, allowing freedom of implementation. Also, allowing the agents to work at the knowledge level means that the system is more robust and can provide intuitive feedback not only for debugging and testing, but also for explanation and justification.

7.5.2 Facilitator Agent

The facilitator agent performed well, never failing to provide communication channels for the other agents and ensuring that service providers were matched to the requestors.

It is proposed that the ability to communicate with other nodes become part of the facilitator's role. This would involve the ability to access the network and interact with facilitators from other user nodes. It would be through the facilitator that other agents would request services from remote systems. For example, solution agents would ask for solutions from other nodes via the facilitator, in much the same way that the facilitator matches service / information requests to suppliers on the local node.

As the facilitator would have access to all information traded between the local and remote nodes, it would also be able to support the monitoring of social / team issues.

7.5.3 User Interface Agent

The current user interface is fit for purpose but suitable only for use by developers. The role of the interface agent remained unchanged throughout the development, and the information portrayed by the agent was proven adequate. The ability to query agents for their current state and interrogate their knowledge was a powerful aid to development. It was originally anticipated that the system would have to perform some degree of source code "beautifying" to ensure that examples and solutions were supplied to the user in their preferred style and format. To do this, the agent would have to learn the user's preference by analysing all supplied code. However, this was rendered obsolete with the advance of IDEs within the last few years. Borland's J Builder now specifies a number of options to which code will be formatted, regardless of its formatting within the file. The formatting is then applied to every source code file displayed. So, the system just needs to be able to provide a

similar set of options for the user to specify their preferences, thus reducing the requirement for learning, and therefore the complexity.

7.5.4 Sensor Agent

The ability of the sensor agent to parse the source code was more than adequate. The way in which the language phrase grammar was simply translated into JESS rules for the agent to use was superb in its simplicity. This leads to the possibility of using the system with languages other than 'C', as the parsing knowledge could be swapped out for a JESS representation of another language grammar.

To work efficiently with another programming language the heuristic within the sensor agent would also have to be adapted to work with the new language. The heuristic allows the system to fragment the source code into statements or constructs, thus reducing the search space for semantic tree generation.

7.5.5 Variable/Function/Expression Agent

The worker agents performed well, their clearly defined goals allowed them to fulfil their roles effectively. The self awareness of their own state which was designed into the agents proved to be invaluable. Without this self awareness the agents were requesting and responding to requests which they were unable to accurately answer. The awareness of their own state ensured that they only responded to requests when they had enough information to do so.

One drawback of developing autonomous agents, which presented itself mostly during the development of the workers, was the inability to debug the system as a whole. It was crucial to the development that information could be ascertained about

an agent's current state - what (if any) requests it had outstanding, what responses it was waiting for, and what its internal knowledge was. All of this information could be requested from each agent, allowing the system to be debugged, although in a fragmented way. A more visual representation of this knowledge should be included in the next iteration of this project; to improve the development's efficiency.

One area where the expression agent was weak was the lack of consistency in the way it translates the semantic tree representation into the pseudo code facts. This needs to be formalised before the system can be moved on. To do this it is felt that the system needs to be able to exploit the semantic tree more. Currently the expression agent only processes the tree right to left at junctions. More intelligent processing is required to allow assertions like, all children of identifiers, on the left of this node, are of type "int". This would give a more accurate representation of the information held within the semantic tree.

7.5.6 Collator Agent

The use of the collator agent to manage the data either from the evaluator or worker agents allowed for a central consistency to the learning side of the system. The use of learning spaces and planes significantly improved the accuracy and confidence in the concepts induced. The ability to add new spaces and planes easily also affords the system some future-proofing.

7.5.7 Detection Agents

A number of improvements should be made in the next iteration of the system, over and above those discussed in section 6.4:

- Further learning spaces and planes need to be added to the system to allow a wider scope of reasoning over code. Module and project level guidelines can then be learnt.
- More relations need to be extracted from the information within the function learning plane.
- By removing specific details from the source code, more generalised rules could be induced. For example, the good practice captured for closing a handle after it has been successfully opened and used could map directly to the freeing of memory after it has been allocated. The ability to move information into meta data is already given, so this would not be a difficult modification.
- The ability to remember the more successful parts of a rule when attempting to relearn one. If a part of a rule has been praised by the user it should have some weight and not be lost or superseded by a rule which, although it incorporates the new information, has a lower confidence rating.

7.5.8 Solution Agent

As discussed in sections 4.3 and 4.4.11, the solution agent was supposed to offer solutions to the defects found. It was initially proposed that the agents would employ case-based reasoning, allowing them to watch good practice and provide snippets of good code from this reservoir of examples. However, it became apparent that this was not required. The information encoded into the rules for each guideline is almost sufficient on its own to inform the user of the error. As the processing is

done at the knowledge level, the semantics of the rule are intuitive. However, to complete the usability of the system, the history of good examples gathered for each guideline in the evaluation agent's long-term memory allows it to supply information, not only about the guideline and its context but also about ways of solving or fixing the defect.

7.6 Overall Performance

The overall architecture was proven to be a solid foundation for the rest of the system. The levels of autonomy within the agents was kept to a manageable level by the use of roles. This, coupled with the ability to debug the system (albeit in a fragmented fashion), allowed easy development and integration of other agents into the system.

The ability of the system to capture the informal benefits of the peer code review and recognise social trends in the team looks promising. The prototype system shows that it will be of use in capturing the individuals preferences, and the simulation shows that the social side of the system is a crucial element to the real benefits and foibles of code review, which may equate to the beneficial emergent behaviour described in earlier chapters.

It is also important to note that the system was able to achieve a good level of learning with real-life small data sets. Compared with other learning systems, this is not a trivial achievement. Learning sets as small as 4 pairs were used, the outcome of which was at least 60% accurate, although this falls below the success criteria (as described in Chapter 5) for the system, and it is felt that this is unusual given the data set size.

Chapter 8

Conclusions & Future Work

This chapter presents the conclusions drawn from this research. Because of its scope, a number of areas are covered by this work. For clarity, the conclusions are therefore presented grouped by the field in which they make an original contribution or enhance knowledge.

8.1 Software Engineering & Quality Assurance

The following conclusions were drawn:

1. The analysis of the peer code review process and the formal software Inspection process lead to the belief that the more informal code review process should not be seen as a substandard Inspection. It has benefits which are complementary to those of Inspection. The process of Inspection

improves the process of finding bugs in software artefacts. The process of peer code review improves the knowledge of the programmers involved and captures the solutions and best practice. It may be contentious but, because prevention is better than cure, peer code review could be more beneficial to improving the quality of source code than Inspection. Indeed, one noted drawback to Gilb's Inspection process is that the process fully describes how to inspect software, but does not detail what to look for.

2. The ability of the code review process to improve knowledge was observed during the team tests as presented in the previous chapters. The ability to teach was evident in the examples supplied by AL and MB during the testing phase, as they both showed noticeable improvement in their programming abilities. AL improved his error handling strategies and understanding of constructs such as the do-while(false) construct, and MB learnt how to correctly check the return values from CreateFile and ReadFile and when to call CloseHandle. Both individuals enjoyed the exercise and realised the benefits of sharing such techniques with more experienced programmers.

3. Another important contribution this work provides is the conclusion that it is possible to capture and exploit best practice using a software system. The prototype, although limited in scope, was able to learn the traits of real world bugs so that it could detect them in other pieces of code. The system was then able to ascertain a consensus among the team on the solution to use to avoid such bugs. This consensus was a representation of the team's best practice.

8.2 Machine Learning

The following conclusion was drawn:

1. Notable about the approach taken within this system was the ability of the system to learn from very small (relative to other machine learning systems) data sets. This was achieved by dividing the original search space into a number of more focused spaces (learning spaces) and using a number of different learning algorithms, each one tailored to learning a particular type of guideline. Learning was then facilitated in each of these learning spaces, with each of these algorithms concurrently. For each learning space the noise was reduced and therefore the accuracy of the concepts increased. Each algorithm was adept at learning certain types of rules, increasing the coverage of the system. This, coupled with direct user feedback / evaluation of the system's performance in applying the induced concepts, ensured that the system remained suitably accurate.

8.3 Agent Technology

The following conclusions were drawn:

1. It was the packaging of the multiple learning algorithms into multiple agents that allowed the system to perform well. The communications and use of other agents such as the collator and the evaluator allowed the system to use softer measures, such as weightings based on the agents' confidence and the perceived confidence of the user. This agent-rule pairing allowed the system to refine the findings of the learning agents and report only those findings in

which it was confident. This extra validation step on the results of the learning mechanism improved the accuracy of the system.

The use of multiple learning mechanisms in one system is not new; George Tecuci presents the disciple architecture in which multiple learning strategies are employed within agents. The disciple architecture combines different learning strategies; a technique for knowledge acquisition is complemented with a technique for machine learning. This is different to the system presented in this thesis, which uses multiple variations of one learning strategy - 'Empirical inductive'. Another clear difference is that in the disciple architecture each system comprises one agent which embodies a complex learning system.

The performance of these systems for a small number (less than 20) of examples was comparable. Demonstrating that the performance of a system that comprises multiple agents embedding many different variants of a single learning strategy, was equivalent to that of a system comprising a single agent embodying a complex learning mechanism, which uses many learning strategies.

There are two main reasons why, for this domain, the approach taken in this system is preferable to that of the disciple architecture. Firstly, the disciple architecture requires a certain amount of prior knowledge. Secondly, the development of the agent would be significantly more work than was necessary for this system, as the learning strategies employed are not trivial.

2. The ability of the system to model the team's emergent behaviour looks promising. The simulations show that a consensus of solutions can be found which represent the team's best practice in the context of individual defects or guidelines, and that the system is able to observe the social structure and influences.

   It is important to note that this system was designed to capture the outputs of the emergent behaviour of the human group. The goal of agents was to model the indefinable but already effective process, not to improve the process.

3. The design of a system which comprises many agents in one node allows for easier development. It has been shown to have improved the performance of the learning mechanism in the prototype, to have achieved results comparable to those captured by more sophisticated learning mechanisms.

8.4 Summary

The main aim of this research was to, "determine if it is feasible to automate best practice; the optimum way to provide this (agents); and the method by which such a system could, a) act as a consultant on best practices and, b) learn the fundamentals of good programming practices."

- The ability to capture best practice using the team as the only source of knowledge is innovative and controversial. There is no mechanism for controlling the team. The system purely captures the consensus between the programmers. If the team's behaviour changes, for example through a

reorganisation of programmers, the best practice could become volatile, reflecting their changing priorities.

- The system finds bugs using the team as a point of reference, capturing best practice through their consensus. The consensus never goes awry; it always tends to follow the safe and correct route – which is the individual aim of all programmers in the team. This assumes that the majority want good coding and a few are experienced.

It has been theorised that a solution to a global problem can emerge from the collective activities of independent agents. Research in this area has demonstrated that highly goal-directed, robust, nearly optimal behaviours can arise from the interactions of simple individual agents [Luger and Stubblefield, 1998]. Luger goes further to say that full intelligence can and does arise from the interactions of many simple, individual, embodied agents.

Although this development is on a much smaller scale, the basic principle still applies - that by having a group of simple (but non-trivial) agents working together with the user, a level of intelligence can emerge that betters the task at hand.

The aim of this system was not to specifically improve software quality, but to mimic the team's emergent behaviour in an attempt to capture the agreed practices which it assumes are the best practice. This work has demonstrated that the prototype system has validated the architecture and that, by adopting an agent oriented approach, the development of a software system to address a less fully defined task can be successful. This approach should prove successful in attempting to capture best practice from an informal social phenomenon - code review.

With this research it was shown that agents can be useful and applicable to real world problems, where traditional software is just not adequate. It is only by the development of this and other practical systems that agents will become accepted by the computer science and software engineering domains as serious software paradigms. Fully generic agent architectures which implement a large range of concepts from BDI to KQML are needed. However, more work is needed to prove agents have potential, that they are able to solve problems not solvable by "normal" software and that they bring new ideas to the field of AI. To do this we need to be able to answer the question, "why agents?" The answer in this case is that we needed to capture the output from a process which we were unable to model or define. A mechanism was needed which would be able to reflect and mimic, without direction, the complex social behaviours within the human process.

8.5 Contributions

This work has contributed to the field of agent technology by showing that an agent based system can be developed which is capable of capturing the emergent behaviour of a team of programmers (8.3.2), resulting in improved code quality (8.1.3) and skills (8.1.2). It has also shown that by applying an agent oriented approach to machine learning, adequate results can be achieved in a real world scenario (8.2.1 and 8.3.1).

8.6 Future work

Ideas for future work include:

- Making the system more situated by integrating it into the Visual C development environment. The system would then be more at hand, so that programmers could more easily request a review of the current source file or project.
- Include agents which wrap the latest version of Lint and Bounds Checker. This would increase the coverage of the system and provide a way of checking for real-time bugs, something the current static analysis framework is unable to support. An extension to this idea is that these agents could include resource management, as the process they encompass can be time consuming and CPU intensive. It would be ideal if the agents were able to schedule reviews / tests whilst the programmer was away from their desk. The agents could then report back when requested by the programmer if the module / function in error is still untouched. They could also ensure that the programmer didn't break the environment for using such tools, ensuring for example, that the configuration files reflected the consensus of the team.
- Improving the processing time by incorporating full mobility and resource management / scheduling into the agents, so that they can move around the network efficiently, exploiting the available resources. This change would need to ensure that only the processing was relocated, and not the preferences and ownership of the agents in question.

- Increase the effectiveness of the learning mechanism by improving the weightings used. This would involve an awareness of the origin of the example / user direction - the level of competence of the supplier. Instead of covering all positive examples, as it does now, the learning mechanism could attempt to cover all the positive examples supplied from peer review, and most of the positive examples ascertained from individuals, such that the concepts used are more generic and may more accurately reflect the group's agreement.

- The learning needs to be improved such that a "wrap on knuckles" for getting one example wrong should not mean the whole rule is discarded when it has previously been correct. A possible suggestion for the agent to limit the change on an error would be to apply weightings to the clauses of a rule, instead of the rule as a whole.

- Expanding language independence. Currently only the sensor agent is designed to support full language independence, with its separate knowledge base. This principle could be applied to the other agents, including porting knowledge and heuristics.

References

[1, ESSI]

Edited by: M. Haug, E. W. Olsen & L. Bergman.
'ESSI: European Systems and Software Initiative'.
Springer, 2001.

[2, IEEE]

IEEE Computer Society.
http://www.computer.org

[3, BCS]

British Computer Society
http://www.bcs.org.uk

[4, ACM]

Associate for computing machinery
http://www.acm.org

[5, Fagan]

M. E. Fagan.
'Design and code inspections to reduce errors in program development'.
Presented: IBM Systems Journal 15, 3 (1976): 182-211.
http://www.research.ibm.com/journal/sj/283/ibmsj2803D.pdf (1988).

[6, Gilb & Graham]

T. Gilb and D. Graham. *"Software Inspection"*
Addison-Wesley Longman, 1993.

[7, NASA]

NASA, SATC (Software Assurance Technology Center).
'Software Formal Inspections'.
http://satc.gsfc.nasa.gov/fi/fipage.html

[8, Arnott]

Sarah Arnott. *'MoD wasted £120m on mismanaged IT'.*
Computing. http://www.computing.co.uk/New/1147382.
November 2003.

[9, BBC]

BBC. *'Computer problems hit CSA payouts'.*
BBC News. http://news.bbc.co.uk/1/hi/uk/3235394.stm".
November 2003.

[10, Arnott]

Sarah Arnott. *'Software causes Prison Server's £7m salary error'.*
Computing. http://www.computing.co.uk/Specials/1131861.
February 2004.

[11, Ranger]

Steve Ranger. *'Chaos as air traffic control fails'.*
Computing. http://www.computing.co.uk/News/1130528.
March 2002.

[12, Computing]

Computing. *'Naivety led to GCHQ IT crisis'.*
Computing. http://www.computing.co.uk/News/1151189.
December 2003.

[13, Arnott]

Sarah Arnott. *'Computing to give evidence on Whitehall IT'.* Computing. http://www.computing.co.uk/News/1152850. February 2004.

[14, Arnott]

Sarah Arnott. *'£1.5bn squandered on government IT'.* Computing. http://www.computing.co.uk/News/1139418. March 2003.

[15, QAI]

Quality Assurance Institute. http://www.qaiusa.com.

[16, Software QA/Test Resource Center]

Software QA/Test Resource Center. http://www.softwareqatest.com.

[17, SEI]

Software Engineering Institute, Carnegie Mellon. http://www.sei.cmu.edu.

[18, SQI]

Software Quality Institute, University of Texas at Austin. http://lifelong.engr.utexas.edu/sqi/index.cfm.

[19, SATC]

Software Assurance Technology Center, NASA. http://satc.gsfc.nasa.gov/homepage.html.

[20, ESI]

European Software Institute. http://www.esi.es.

[21, Laitenberger]

Oliver Laitenberger. *'Studying the Effects of Code Inspection and Structural Testing of Software Quality'.*

[22, Laitenberger]

Oliver Laitenberger.
'A Survey of Software Inspection Technologies'.

[23, Butler & Johnson]

R. W. Butler & S. C. Johnson. *'Formal methods for life-critical software'.*
NASA Langley Research Center.
http://archive.larc.nasa.gov/shemesh/paper-cia/

[24, ERCIM]

ERCIM Working group on Formal Methods for Industrial Critical Systems.
http://www.iniralpes.fr/vasy/fmics

[25, McCorduck]

McCorduck.
Presented in *'Artificial Intelligence: Structure and Strategies for Complex Problem Solving'*, George F. Luger & William A. Stubblefield.
3rd Edition, 1998. Addison Wesley.

[26, Porter et al]

A. A. Porter, H. P. Siy, C. A. Toman & L. G. Votta.
'An Experiment to Assess the Cost-Benefit of Code Inspections in Large Scale Software Development'
Presented in *'IEE Transactions on Software Engineering'*, Vol 23. No 6.
June 1997.

[27, Pressman]

Roger Pressman. *'Software engineering: A practitioner's approach'.*
Adapted by Darrel Ince.
Fifth edition, 2000. McGraw-Hill International (UK) Limited.

[28, Humphrey]

Watts. S. Humphrey. *'Some Programming Principles: People'.*
http://interactive.sei.cmu.edu/news@sei/columns/watts_new/watts-new.pdf

[29, Thibodeau]

Patrick Thibodeau. *'Future Watch: Software Bugs on the March'.*
http://www.computerworld.com/softwaretopics/software/story/0,10801,86403,00.html
October 2003.

[30, Newscientist]

Newscientist.
http://www.newscientist.com/news/news.jsp?id=ns99992757
September 2002.

[31, Jackson]

M. Jackson. Abstract: *'Why software writing is difficult and will remain so'.*
ACM digital library.
http://portal.acm.org/

[32, Luger & Stubblefield]

George F. Luger & William A. Stubblefield.
'Artificial Intelligence: Structure and Strategies for Complex Problem Solving', P17,
Addison-Wesley, 1998.

[33, Maes]

Pattie Maes. *'Designing Autonomous Agents: Theory and practice from Biology to Engineering and Back'*, MIT Press, 1990.
Presented in *'Artificial Intelligence: A New Synthesis'*, N. Nilsson, p7.
Morgan Kaufmann, 1998.

[34, Kin & Lerch]

J. Kim & F. J. Lerch.
'Why Is Programming (Sometimes) So Difficult? Programming as Scientific Discovery in Multiple Problem Spaces'.
October 1996.

[35, Walenstein]

A. Walenstein.
'Cognitive Support in Software Engineering Tools: A Distributed Cognition Framework'.
2002.

[36, IEEE]

IEEE. *'IEEE Standard for Software Reviews and Audits'.*
ANSI/IEEE Std. 1028-1988.

[37, Jennings and Wooldridge]

Nicholas R. Jennings & Michael Wooldridge.
'Agent-Oriented Software Engineering'.

[38, Maes]

Pattie Maes. *'Agents that Reduce Work and Information Overload'.*
Presented in *'Software Agents'*, Ed: Jeffery M. Bradshaw, p149.
AAAI Press/The MIT Press, 1997.

[39, Brenner, Zarnekow & Wittig]

Walter Brenner, Rudiger Zarnekow & Hartmut Wittig.
'Intelligent Software Agents'.
Springer, 1998.

[40, Carnegie Mellon University]

'Pleiades'. The Intelligent Software Agents Lab, The Robotics Institute, Carnegie Mellon University.
http://www-2.cs.cmu.edu/~softagents/resina_agent_arch.html
2001.

[41, Carnegie Mellon University]

'The RETSINA Agent Architecture'. The Intelligent Software Agents Lab, The Robotics Institute, Carnegie Mellon University.
http://www-2.cs.cmu.edu/~softagents/resina_agent_arch.html
2001.

[42, Malone, Lai & Grant]

Thomas W. Malone, Kum-Yew Lai & Kenneth R. Grant.
'Agents for Information Sharing and Coordination: A History and Some Reflections'.
Presented in *'Software Agents'*, Ed: Jeffery M. Bradshaw, p110.
AAAI Press/The MIT Press, 1997.

[43, Pressman]

Pressman, R., S., *'Software Engineering: Practitioners Approach'.*
McGraw-Hill, 1992.

[44, Pomberger]

Pomberger, G., *'Software Engineering and Modula-2'.*
Prentice-Hall, 1984.

[45, Perrolle]

Perrolle, J., A.,
'Surveillance and Privacy in Computer Supported Cooperative Work', in New Technology, in Surveillance and Social Control, David Lyon and Elia Zureik, eds.
University of Minnesota Press, 1995.
http://www.ccs.neu.edu/home/perrolle/privacy.html.

[46, Zin & Foxley]

Zin, A., M., & Foxley, E., 'Automatic Program Assessment System'.
http://www.cs.nott.ac.uk/~ceilidh/papers/ASQA.html.

[47, Salmon]

Ayse Salmon's Teaching assistant.
Unpublished PhD Thesis, OBU, 2001.

[48, Lieberman]

Henry Lieberman. *'Interfaces That Give and Take Advice'*.
http://web.media.mit.edu/~lieber/Lieberary/Advice/Advice.html.

[49, Morris & Maglio]

Joan Morris & Paul P. Maglio.
'When Buying On-line, Does Price Really Matter?'
http://web.media.mit.edu/~joanie/sardine/chi-pricematters-shortpaper.pdf.

[50, Shearin & Lieberman]

Sybil Shearin & Henry Lieberman.
'Intelligent Profiling by Example'.
http://www.media.mit.edu/~sibyl/projects/apt/iui2001_profile.pdf.

[51, Kumar et al]

A. Kumar, S. C. Sundararajan & H. Lieberman.
'Common Sense Investing: Bridging the Gap Between Expert and Novice'.
http://portal.acm.org/citation.cfm?id=986015.

[52, Luck et al]

Micheal Luck, Peter McBurney & Chris Preist.
'Agent Technology: Enabling Next Generation Computing. A Roadmap for Agent Based Computing'.
January 2003.

[53, Sheth]

B. D. Sheth. *'A Learning Approach to Personalised Information Filtering'*.
Masters Thesis, MIT.
February 1994.

[54, Payne & Edwards]

T. R. Payne & P. Edwards.
"Interface Agents that Learn: An Investigation of Learning Issues in a Mail Agent Interface".
October 1995.

[55, PC-Lint]

PCLint for C/C++.
http://www.gimpel software.com/

[56, LCLint]

LCLint University of Virginia, *'LCLint Homepage'*.
http://lclint.cs.virginia.edu/.

[57, NuMega]

Compuware/NuMega, *'Code Review for Visual Basic'*,
http://www.compuware.com/products/numega/dps/vb/cr.htm.

[58, Parasoft]

Code Wizard, Parasoft.
http://www.parasoft.com/.

[59, Review Pro people (StdCorp)]

ReviewPro, Std Corp.
http://www.stdcorp.com/

[60, Hills]

Chris Hills. *'MISRA-C Compliance Matrix using PC-Lint'*.
December 2001.

[61, MISRA]

MISRA Guidelines for the user of the C Language in Vehicle Based Software.
http://www.misra.org.uk/

[62, Meterns et al]

P. Meterns, F. Bodendorf, W. König, A. Picot & M. Schumann.
'Grundzuge der Wirtschaftsinformatik'.
Presented in: *'Intelligent Software Agents'.*
Walter Brenner, Rudiger Zarnekow & Hartmut Wittig.
Springer, 1998.

[63, Maes]

P. Maes. *'Agents that reduce work and information overload'.*
Presented in: *'Intelligent Software Agents'*, Walter Brenner, Rudiger Zarnekow & Hartmut Wittig, p258.
Springer, 1998.

[64, Friedman-Hill]

Friedman-Hill, E., J., 'Jess Homepage',
http://herzberg.ca.sandia.gov/jess/.

[65, Mercer & Greenwood]

S. Mercer & S. Greenwood.
'A Multi-agent Architecture for Knowledge Sharing'.
Proceedings of the Sixteenth European Meeting on Cybernetics and Systems Research, 2001.

[66, Finin]

T. Finin. *'Tutorial on Agent Communication Languages'.*
Autonomous Agents 2001, Montreal, Canada, May 2001.

[67, Wooldridge & Parsons]

M. Wooldridge & S. Parsons.
'Tutorial on Rational Action in Autonomous Agents'.
Autonomous Agents 2001, Montreal, Canada, May 2001.

[68, Kolodner]

J. Kolodner. *'Case-based reasoning'.*
Morgan, Kaufmann, 1993.

[69, Leake]

D. B. Leake.
'Case-based reasoning: Experiences, Lessons & Future Directions".
AAAI, 1996.

[70, Ossowski et al]

S. Ossowski, J. Pérez-de-la-Cruz, J. Z. Hernández, J.M. Maseda, A. Fernández, M. V. Belmonte, A García-Serrano, J. M. Serrano, R. León and F Carbone.
'Towards a Generic Multiagent Model for Decision Support: Two Case Studies'.
Presented in the Proceedings of The First European Workshop on Multi-Agent Systems. Eds: M. d'Inverno, C Sierra, F. Zambonelli.
December 2003.

[71, FIPA-OS]

FIPA Foundation for Intelligent Physical Agents.
'FIPA-OS Tutorial Step 4: Using the JESS Agent'.
http://fipa-os.sourceforge.net/
August, 2000.

[72, Kennion]

Tarmel Kennion.
'A Community of Expert-System Agents'.

[73, Finin & Labrou]

Finin, T., & Labrou, Y.,
'Specification of the KQML Agent-Communication Language'.
http://www.cs.umbc.edu/kqml/kqmlspec/spec.html.

[74, Genesereth]

M. R. Genesereth. *'An Agent-Based Framework for Interoperability'.*
Presented in *'Software Agents'*, Chapter 15, Ed: J. M. Bradshaw.
AAAI Press / The MIT Press, 1997.

[75, Finin, Labrou & Mayfield]

T. Finin, Y. Labrou & J. Mayfield.
'KQML as an Agent Communication Language'.
Presented in Bradshaw, p305.

[76, Luger & Stubblefield]

G. F. Luger & W. A. Stubblefield. *'Artificial Intelligence'.*
Addison Wesley, 3rd Edition, p 34.

[77, Skvarcius & Robinson]

Romualdas Skvarcius & William B. Robinson.
'Discrete Mathematics with Computer Science Applications', p197.
The Benjamin / Cummings Publishing Company, Inc. 1986.

[78, Rao/Georgeff]

A. S. Rao & M. P. Georgeff. *'BDI Agents: From Theory to Practice'.*
Proceedings of the First International Conference on Multi-Agent-Systems (ICMAS), San Francisco, 1995.

[79, Brenner]

Walter Brenner, Rudiger Zarnekow & Hartmut Wittig.
'Intelligent Software Agents', p70.
Springer, 1998.

[80, Maes]

P. Maes.
'Modeling Adaptive Autonomous Agents'.
http://citeseer.ist.psu.edu/maes94modeling.html

[81, Luger & Stubblefield]

G F Luger and W A Stubblefield, *'Artificial Intelligence'*.
Addison-Wesley, 1998, p603.

[82, Caruana & Freitag]

R Caruana & D Freitag, 'Greedy Attribute Selection'.
Carnegie Mellon University, USA.

[83, Luger & Stubblefield]

G F Luger and W A Stubblefield, *'Artificial Intelligence'*.
Addison-Wesley, 1998.

[84, Perugini et al]

D. Perugini, S. Wark, A. Zschorn, D. Lambert, L. Sterling & A. Pearce.
'Agents in Logistics Planning – Experiecnes with the Coalition Agents Experiemtn Project'.
Presented at Workshop 5 "Agents at work".
AAMAS 2004, Melbourne.

[85, Connell et al]

R. Connell, F. Lui, D. Jarvis & M. Watson.
'The Mapping of Courses of Action Derived from Cognitive Work Analysis to Agent Behaviours'.
Presented at Workshop 5 "Agents at work".
AAMAS 2004, Melbourne.

[86, Koller & Sahami]

D Koller and M Sahami, *'Toward Optimal Feature Selection'.*
Proceedings of the Thirteenth International Conference.
Morgan Kaufmann, 1996.

[87, Raman & Ioerger]

B Raman and T R Ioerger, *'Instance Based Filter for Feature Selection'*,
Texas A&M University, USA.

[88, Talavera]

Luis Talavera,
'Dynamic Local Feature Selection in Incremental Clustering'.
Universitat Politecnica de Catalunya, Spain.

[89, Raman and Ioerger]

B Raman and T R Ioerger,
'Enhancing Learning using Feature and Example selection'.
Texas A&M University, USA.

[90, HMGCC]

S Mercer & P Denison, *'HMGCC Global Coding Standards'.*
HMGCC, 2001.

[91, Russell & Norvig]

S, Russell & P. Norvig.
'Artificial Intelligence: A Modern Approach', p526.
1st Edition, Prentice Hall, 1995.

[92, Wetzel]

B Wetzel. *'Project: Selection Engine'.*
http://sourceforge.net/projects/selectionengine/.

[93, Nilsson]

N. Nilsson, *'Artificial Intelligence: A New Synthesis'*.
Morgan Kaufmann, 1998.

[94, Quinlan]

J R Quinlan. *'Learning Logical Definitions from Relations'*.
Machine Learning. Morgan Kaufmann, 1990.

[95, Califf & Mooney]

M. E. Califf & R. J. Mooney.
'Relational Learning of Pattern-Match Rules for Information Extraction'.

[96, Craven & Slattery]

M. Craven & S. Slattery.
'Relational Learning with Statistical Predicate Invention: Better Models for Hypertext'.

[97, Pearce et al]

A. R. Pearce, T. Caelli & W. F. Bischof.
'Learning Relational Structures: Applications in Computer Vision'.

[98, Pearce]

A. R. Pearce.
PhD These: *'Relational evidence theory and spatial interpretation procedures'*.
September, 1996.

[99, Perugini et al]

D. Perugini, S. Wark, A. Zschorn, D. Lambert, Leon Sterling & A. Pearce.
'Agents in Logistics Planning – Experiences with Coalition Agents Experiment Project'.
Presented at Workshop 5: "Agents at work" part of AAMAS 2003, Melbourne.

[100, Pearce]

A. R. Pearce, T. Caelli & W. F. Bischof.
'Efficient Spatial and Temporal Learning Procedures and Relational Evidence Theory'.

Appendices

Tests

Test1

```
/*
* MyGetFileSize
*
* Preconditions:
* szFilename is valid (i.e. contains path to file to
open
* and is null terminated).
*
* Postconditions:
* returns -1 on failure, or size in bytes of file on
success.
*
* PS: don't worry about the ... - I know it wont
compile.. but it shouldn't
*     be important - and I cant remember the params for
CreateFile!
*/
int MyGetFileSize(char *szString)
{
  HANDLE hFile;
  DWORD dwSize=0;

  hFile=CreateFile(szString, ...);
  dwSize=GetFileSize(hFile, NULL);

  return dwSize;
}


/*
Problems:
      The return value from CreateFile is not checked.
      The return value from GetFileSize is not checked.
      (close handle).
*/
```

Test2

```
/*
* GetFileHeader
*
* for some reason the file starts with a two digit
number, this function
* returns that number as a DWORD.
*
* Preconditions:
* szFilename is valid (i.e. contains path to file to
open
* and is null terminated).
*
* Postconditions:
* returns 0 on failure, or the header on success.
*
*/
DWORD GetFileHeader(char *szFilename)
{
  HANDLE hFile;
  DWORD dwHeader=0, dwRead;
  char lpBuffer[3];

  hFile=CreateFile(szFilename, GENERIC_READ, 0, NULL,
OPEN_EXISTING, 0, NULL);
  ReadFile(hFile, lpBuffer, 2, &dwRead, NULL);
  dwHeader=strtoul(lpBuffer, NULL, 10);
  return dwHeader;
}

/*
Problems:
      The return value from CreateFile is not checked.
      The return value from ReadFile is not checked.
      The return value from Strtoul is not sanity
checked.
      The handle to the file is not closed.
*/
```

Test3

```
/*
* myWriteHeader, writes a number as a string to a file.
*
*returns TRUE.if it worked, FALSE on error.
*/
BOOL myWriteHeader(char *szFilename, DWORD dwHeader)
{
      HANDLE hFile=INVALID_HANDLE_VALUE;
      char szString[32];
      DWORD dwWritten=0;

      debug("I'm about to open a file!");
      hFile=CreateFile(szFilename);
      wsprintf(szString, "%lu", dwHeader);
      WriteFile(hFile, szString, strlen(szString)+1,
&dwWritten, NULL);
      return TRUE;
}

/*
Problems:
      The file handle is not closed.
      The return values from CreateFile and WriteFile
are not checked.
*/
```

Test4

```
/*
* this function reads in a number from a file, works
that out as a percentage of the passed in total, and
writes back the result to the file.
*
*/
BOOL myUpdateHeader(char *filename, int total)
{
      HANDLE file=INVALID_HANDLE_VALUE;
      DWORD Count, read, written;
      char buffer[5], string[32];
      float f, percent;
      DWORD towrite;

      file=CreateFile(filename);
      ReadFile(hFile, buffer, 2, &read, NULL);
      Count=stroul(buffer, NULL, 10);

      f=(float)Count/(float)total;
      percent=f*100;
      towrite=(DWORD)percent;

      wsprintf(string, "%lu", towrite);
      WriteFile(file, string, lstrlen(string)+1,
&written, NULL);
      return TRUE;
}


/*
Problems:
      variable naming is inconsistent!
      return values are not checked from Create/Read and
WriteFile
      file is not closed at return points.
*/
```

Test5

```
BOOL myBlankHeader(char *szFilename)
{
      HANDLE handle=INVALID_HANDLE_VALUE;
      DWORD Read, Written;
      int loop;
      char Buffer[33];

      handle=CreateFile(szFilename);
      ReadFile(handle, Buffer, 32, &Read, NULL);

      for (loop=0; loop<32; loop++)
      {
            if ((Buffer[loop]>='0') &&
(Buffer[loop]<='9'))
            {
                  Buffer[loop]='*';
            }
      }

      SetFilePointer(handle, 0, 0, FILE_BEGIN);
      WriteFile(handle, Buffer, loop, &Written, NULL);
      return TRUE;
}

/*
Problems:
      inconsistent use of variable naming.
      no close handle
      no return value checks
*/
```

Test6

```
BOOL myGetAndRemoveHeader(char *szFilename)
{
      HANDLE hFile=INVALID_HANLDE_VALUE;
      DWORD dwRead, dwWritten, dwSize;
      char *lpBuffer, *lpPtr;

      do
      {
            hFile=CreateFile(szFilename);
            dwSize=GetFileSize(hFile, NULL);
            lpBuffer=(char*)malloc(dwSize+1);

            ReadFile(hFIle, lpBuffer, dwSize, &dwRead,
NULL);
            lpBuffer[dwSize]='\0';

            lpPtr=lpBuffer;
            while (lpPtr!=NULL)
            {
                  if ((*lpPtr>='0') && (*lpPtr<=9))
                  {
                        lpPtr++;
                  }
                  else
                  {
                        break;
                  }
            }

            SetFilePointer(hFile, 0, 0, FILE_BEGIN);
            bReturn=WriteFile(hFile, lpPtr,
lstrlen(lpPtr)+1, &dwWritten, NULL);
            if (bReturn!=TRUE)
            {
                  break;
            }

            free(lpBuffer);
            return TRUE;
      }
      while (FALSE);
      return FALSE;
}

/*
Problems:
      Return values (you know the drill by now!)
      file isn't closed.
*/
```

Test7

```
BOOL myExtractAndWriteHeader(char *szFilename, char
*szOutputFile)
{
      HANDLE hFile=INVALID_HANLDE_VALUE;
      DWORD dwRead, dwWritten, dwSize;
      char *lpBuffer, *lpPtr;

      hFile=CreateFile(szFilename);
      if (hFile!=INVALID_HANDLE_VALUE)
      {
            debug("file wont open!);
            return FALSE;
      }

      dwSize=GetFileSize(hFile, NULL);
      lpBuffer=(char*)malloc(dwSize+1);

      ReadFile(hFIle, lpBuffer, dwSize, &dwRead, NULL);
      lpBuffer[dwSize]='\0';

      lpPtr=lpBuffer;
      while (lpPtr!=NULL)
      {
            if ((*lpPtr>='0') && (*lpPtr<=9))
            {
                  lpPtr++;
            }
            else
            {
                  break;
            }
      }
      *lpPtr='\0';

      hFile=CreateFile(szOutputFile);
      if (hFile==INVALID_HANDLE_VALUE)
      {
            debug("unable to open second file!");
            return FALSE;
      }

      WriteFile(hFile, lpPtr, lstrlen(lpPtr)+1,
&dwWritten, NULL);
      free(lpBuffer);
      return TRUE;
}

/*
Problems:
      Error handling is rubbish (checks missing, check
on first CreateFile is incorrect).
      memory is not always freed, file handles are not
closed.
*/
```

Solutions for Test 2

Solution 1

Author PA.

```
#define FILE_NAME "test2.c"
#define BUFFER_SIZE 3
#define BYTES_TO_READ 2
#define BASE_NUMBER 10


/*
* test2GetFileHeader (Pete A)
*
* for some reason the file starts with a two digit
number, this function
* returns that number as a DWORD.
*
* Preconditions:
* szFilename is valid (i.e. contains path to file to
open
* and is null terminated).
*
* Postconditions:
* returns 0 on failure, or the header on success.
*
*/
DWORD test2GetFileHeader(char *szFilename)
{
      HANDLE      hFile = INVALID_HANDLE_VALUE;
      DWORD dwHeader = 0;
      DWORD dwRead = 0;
      BYTE  lpBuffer[BUFFER_SIZE] = {0};

      do
      {
            // Check pointer parameter
            if(szFileName == NULL)
            {
                  _RPT1(_CRT_WARN,"%s -
test2GetFileHeader() szFileName == NULL.\n",
                                    FILE_NAME);
                  break;
            }

            // Open file for Reading, only if it exists
            hFile = CreateFile((LPCTSTR)szFilename,

      GENERIC_READ,
```

```
                                                                    0,

      NULL,

      OPEN_EXISTING,
                                                                    0,

      NULL);

            // Check the file was successfully opened
            if(hFile == INVALID_HANDLE_VALUE)
            {
                  _RPT2(_CRT_WARN,"%s -
test2GetFileHeader()::CreateFile() %d\n",
                              FILE_NAME,GetLastError());
                  break;
            }

            // Read the file into a buffer
            if(!ReadFile(hFile,
                              (LPVOID)lpBuffer,
                              BYTES_TO_READ,
                              &dwRead,
                              NULL))
            {
                  _RPT2(_CRT_WARN,"%s -
test2GetFileHeader()::ReadFile() %d\n",
                              FILE_NAME,GetLastError());
                  break;
            }

            // Check that we have read the first two
digits
            if(dwRead != BYTES_TO_READ)
            {
                  _RPT2(_CRT_WARN,"%s -
test2GetFileHeader() BYTES_TO_READ != %d\n",
                              FILE_NAME,dwRead);
                  break;
            }

            // Convert string to unsigned long
            dwHeader = (DWORD)strtoul((const
char*)lpBuffer,

                  NULL,

                  BASE_NUMBER);

            if((dwHeader == LONG_MAX) || (dwHeader ==
LONG_MIN))
            {
                  _RPT2(_CRT_WARN,"%s -
test2GetFileHeader()::strtoul() "
                  "LONG_MAX or LONG_MIN
%d\n",FILE_NAME,dwHeader);
                  // Set to zero for failure
```

```
                    dwHeader = 0;
                    break;
                }

        } while(0);

        // Close the file handle if it was opened
        if(hFile != INVALID_HANDLE_VALUE)
        {
                (void)CloseHandle(hFile);
        }

        return dwHeader;
}
```

Solution 2

Author JA.

```
DWORD GetFileHeader(char *szFilename)
{
  HANDLE hFile=INVALID_HANDLE_VALUE ;
  DWORD dwHeader=0, dwRead;
  char lpBuffer[3];

  do
    {

        hFile=CreateFile(szFilename, GENERIC_READ, 0,
NULL, OPEN_EXISTING, 0, NULL);
        if (hFile==INVALID_HANDLE_VALUE)
    {
      _RPT1(_CRT_WARN,"GetFileHeader: CreateFile failed
with errorcode %d ",GetLastError());
            break;
    }

       if ( ReadFile(hFile, lpBuffer, 2, &dwRead, NULL)
==0 )
        {
           _RPT1(_CRT_WARN,"GetFileHeader: ReadFile
failed with errorcode %d ",GetLastError());
                break;
       }

        dwHeader=strtoul(lpBuffer, NULL, 10);
         if (dwHeader==0)
    {
      _RPT0 (_CRT_WARN,"GetFileHeader: strtoul
failed");
    }

} while (FALSE);

if (hFile!=INVALID_HANDLE_VALUE)
{
    if (CloseHandle(hFile)==0)
    {
        // Not a lot we can do but tell the user anyway
        _RPT1(_CRT_WARN,"GetFileHeader:
CloseHandlefailed with errorcode %d ",GetLastError());
       }
}

  return dwHeader;
}
```

Solution 3

Author AGo.

```
DWORD GetFileHeader(char *szFilename)
{
  HANDLE hFile = INVALID_HANDLE_VALUE;
  DWORD dwHeader = 0,
  DWORD dwRead = 0;
  char lpBuffer[3] = { 0x00 };
  DWORD dwRetVal = 0;
  const int nReadSize = 2;

  do
  {
    hFile = CreateFile(szFilename, GENERIC_READ, 0,
NULL, OPEN_EXISTING, 0, NULL);
    if(INVALID_HANDLE_VALUE == hFile)
    {
      /* Didn't open the file */
      break;
    }

    if( (!ReadFile(hFile, lpBuffer, nReadSize, &dwRead,
NULL)) && (nReadSize != dwRead) )
    {
      /* Didn't read correct number of bytes */
      break;
    }

    dwHeader=strtoul(lpBuffer, NULL, 10);
    if( (ULONG_MAX == dwHeader) || (0 == dwHeader) )
    {
      /* Can't convert */
      break;
    }

    /* We don't check for overflow as a DWORD is 4 bytes
and we're reading 2 */

    /* Success */
    dwRetVal = dwHeader;
  } while (FALSE);

  /* Tidy up */
  if(INVALID_HANDLE_VALUE != hFile)
  {
    CloseHandle(hFile);
    hFile = INVALID_HANDLE_VALUE;
  }

  return dwRetVal;
}
```

Solution 4

Author AGi.

```
#define NUMBER_OF_HEADER_DIGITS     2
#define BASE_OF_HEADER              10

/*
* GetFileHeader
*
* for some reason the file starts with a two digit
number, this function
* returns that number as a DWORD.
*
* Preconditions:
* szFilename is valid (i.e. contains path to file to
open
* and is null terminated).
*
* Postconditions:
* returns 0 on failure, or the header on success.
*
*/

DWORD GetFileHeader( char *szFilename )
{
      HANDLE hFile            = INVALID_HANDLE_VALUE;
      DWORD dwHeader          = 0,
            dwRead            = 0,
            dwTempResult      = 0;

                        // +1 for the NULL terminator
      char lpBuffer[NUMBER_OF_HEADER_DIGITS + 1] = {0};

      do {

            hFile = CreateFile( szFilename,
                                GENERIC_READ,
                                FILE_SHARE_READ,
                                NULL,
                                OPEN_EXISTING,

      FILE_ATTRIBUTE_NORMAL,
                                NULL );

            if( hFile == INVALID_HANDLE_VALUE ) {

                  // Handle error
                  break;
            }

            if( !ReadFile( hFile,
                           lpBuffer,
                           NUMBER_OF_HEADER_DIGITS,
                           &dwRead,
```

```
                                NULL) ) {

                    // Handle error
                    break;
              }

              if( dwRead != NUMBER_OF_HEADER_DIGITS ) {

                    // Handle error
                    break;
              }

              dwTempResult = strtoul( lpBuffer, NULL,
BASE_OF_HEADER );

              if( ( dwTempResult == 0 ) ||
                    ( dwTempResult == LONG_MAX ) ||
                    ( dwTempResult == LONG_MIN ) ) {

                    // Handle error
                    break;
              }

              // Set the return value
              dwHeader = dwTempResult;
        }
        while( FALSE );

        if( hFile != INVALID_HANDLE_VALUE ) {

              CloseHandle( hFile );
              hFile = NULL;
        }

        return dwHeader;
}
```

Solution 5

Author PB.

```
#define BUFF_SIZE (2)                  // 2 digit number
#define MAX_NUMBER (99)                // Maximum two digit
decimal number

/*
* GetFileHeader
*
* for some reason the file starts with a two digit
number, this function
* returns that number as a DWORD.
*
* Preconditions:
* szFilename is valid (i.e. contains path to file to
open
* and is null terminated).
*
* Postconditions:
* returns 0 on failure, or the header on success.
*
*/
DWORD GetFileHeader(char *szFilename)
{
    HANDLE  hFile    = INVALID_HANDLE_VALUE;
    DWORD   dwHeader = 0,
            dwRead   = 0,
            dwRet;
    char    lpBuffer[BUFF_SIZE + 1];        // Add 1 for
NULL terminator

    hFile = CreateFile(szFilename, GENERIC_READ, 0,
NULL, OPEN_EXISTING, 0, NULL);

    if (hFile != INVALID_HANDLE_VALUE)
    {
        dwRet = ReadFile(hFile, lpBuffer, BUFF_SIZE,
&dwRead, NULL);

        CloseHandle(hFile);                 // Done with
this now

        if ( (dwRet) && (dwRead == BUFF_SIZE) ) // Check
for successful read
        {
            lpBuffer[BUFF_SIZE] = '\0';     // Add NULL
terminator
            dwHeader=strtoul(lpBuffer, NULL, 10); //
Read decimal number
            if ( dwHeader > MAX_NUMBER )    // Check
range
            {
               dwHeader = 0;
```

```
                }
            }
        }
        return dwHeader;
}
```

Solution 6

Author PT.

```
#define HEADER_LENGTH    2    // Number of digits in
header

/*
* GetFileHeader
*
* for some reason the file starts with a two digit
number, this function
* returns that number as a DWORD.
*
* Preconditions:
* szFilename is valid (i.e. contains path to file to
open
* and is null terminated).
*
* Postconditions:
* returns 0 on failure, or the header on success.
*
*/
DWORD GetFileHeader(char *szFilename)
{
    HANDLE  hFile                          =
INVALID_HANDLE_VALUE;
    DWORD   dwHeader                       = 0;    // post
condition
    DWORD   dwRead                         = 0;
    char    szBuffer[HEADER_LENGTH+1]      = {0};  // +1
for zero term

    do
    {
        // Open the file
        hFile=CreateFile(szFilename, GENERIC_READ, 0,
NULL, OPEN_EXISTING, 0, NULL);
        if (hFile == INVALID_HANDLE_VALUE)
        {
            _RPT2(  _CRT_WARN,"GetFileHeader: Failed to
open file '%s', error %lu\n",
                    szFileName,GetLastError());
            break;
        }

        // read the header
        if  (   (!ReadFile(hFile, szBuffer,
HEADER_LENGTH, &dwRead, NULL))
            &&  (dwRead != HEADER_LENGTH)
            )
        {
            _RPT4(  _CRT_WARN,"GetFileHeader: Failed to
read file '%s', read %lu of %lu, error %lu\n",
```

```
szFileName,dwRead,HEADER_LENGTH,GetLastError());
            break;
        }

        // covert the header
        //     Zero termination is handled by pre-
initing buffer to all zeros)
        dwHeader=strtoul(szBuffer, NULL, 10);
        if (dwHeader==0)
        {
            _RPT2(  _CRT_WARN,"GetFileHeader: Possble
header conversion error on file '%s', header was
'%s'\n",
                    szFileName,szBuffer);
            //     strtoul returns zero on failure,
satisfying postcondition so
            //     no need to reset value to zero
            break;
        }

        // success
    }
    while(0);

    // cleanup
    if (hFile != INVALID_HANDLE_VALUE)
    {
        CloseHandle(hFile);
        hFile = INVALID_HANDLE_VALUE;
    }

    return dwHeader;
}
```

Solution 7

Author MF.

```
#define    HEADER_SIZE    2

DWORD GetFileHeader(char *szFilename)
{
  HANDLE hFile  = INVALID_HANDLE_VALUE ;
  DWORD dwHeader=0, dwRead  = 0;
  char lpBuffer[HEADER_SIZE+1];

  do
  {
        if ((hFile=CreateFile(szFilename, GENERIC_READ,
0, NULL,   OPEN_EXISTING, 0, NULL)) ==
INVALID_HANDLE_VALUE)
        {
            break;
        };
        if (!ReadFile(hFile, lpBuffer,  HEADER_SIZE,
&dwRead, NULL) )
        {
            break;
        };
        if (dwRead != HEADER_SIZE)
        {
            break;
        };
        lpBuffer[HEADER_SIZE] = '\0';
        if ((dwHeader=strtoul(lpBuffer, NULL, 10) ) ==
0)
        {
            // Report an error if you want, but keep the
assignment as a failure return
        };
  } while (0);
  if (hFile != INVALID_HANDLE_VALUE)
  {
      CloseHandle(hFile);            // I know, the ret
value isn't checked :-)
      hFile = INVALID_HANDLE_VALUE;
  };
  return dwHeader;
}
```

Solutions for Test 5

Solution 1

Author PA

```
#define BUFFER_SIZE 33
#define BYTES_TO_READ 32

BOOL myBlankHeader(char *szFilename)
{
      HANDLE      hFile=INVALID_HANDLE_VALUE;
      DWORD dwRead = 0;
      DWORD dwWritten = 0;
      DWORD dwLoop = 0;
      char  szBuffer[BUFFER_SIZE] = {0};
      BOOL  bRet = FALSE;
      DWORD dwSetRet = 0;
      BOOL  bWriteRet = FALSE;

      do
      {
            // Check for null pointer
            if(szFilename == NULL)
            {
                  debug("szFilename == NULL");
                  break;
            }

            // Open file!
            hFile=CreateFile((LPCTSTR)szFilename);

            // check file handle
            if(hFile == INVALID_HANDLE_VALUE)
            {
                  debug("CreateFile Failed");
                  debug(GetLastError());
                  break;
            }

            // Read the file contents
            if(!ReadFile(hFile,
                              (LPVOID)szBuffer,
                              BYTES_TO_READ,
                              &dwRead,
                              NULL))
            {
                  debug("ReadFile Failed");
                  debug(GetLastError());
                  break;
            }
```

```
            // Check that we have read the correct
number of bytes
            if(dwRead != BYTES_TO_READ)
            {
                  debug("dwRead != BYTES_TO_READ");
                  break;
            }

            // Mercer
            for (dwLoop=0; dwLoop<BYTES_TO_READ;
++dwLoop)
            {
                  if ((szBuffer[dwLoop]>='0') &&
(szBuffer[dwLoop]<='9'))
                  {
                        szBuffer[dwLoop]='*';
                  }
            }

            // Point to beginning
            dwSetRet = SetFilePointer(    hFile,
                                                      0,

      NULL,

      FILE_BEGIN);

            // Check SetFilePointer() return value
            if(dwSetRet == INVALID_SET_FILE_POINTER)
            {
                  debug("SetFilePointer() fail
INVALID_SET_FILE_POINTER");
                  debug(GetLastError());
                  break;
            }

            // Write data to file
            bWriteRet = WriteFile(hFile,

      (LPCVOID)szBuffer,
                                          dwLoop,
                                          &dwWritten,
                                          NULL);

            // Check return value and bytes written
            if((!bWriteRet) || (dwLoop != dwWritten))
            {
                  debug("WriteFile() fail, or dwLoop !=
dwWritten");
                  debug(GetLastError());
                  break;
            }

            bRet = TRUE;

      } while(0);
```

```
        // Close Handle
        if(hFile != INVALID_HANDLE_VALUE)
        {
                (void)CloseHandle(hFile);
        }

        return bRet;
}
```

Solution 2

Author JA.

```
BOOL myBlankHeader(char *szFilename)
{
      HANDLE hHandle=INVALID_HANDLE_VALUE;
      DWORD dwRead, dwWritten;
      int iLoop;
      char szBuffer[33];
      BOOL bRet=FALSE;

      do
      {

            hHandle=CreateFile(szFilename);
            if (hHandle==INVALID_HANDLE_VALUE)
            {
                        _RPT1(_CRT_WARN,"myBlankHeader:
CreateFile failed with errorcode %d ",GetLastError());
                  break;
            }

            if ( ReadFile(hHandle, dwBuffer, 32,
&dwRead, NULL)  ==0 )
            {
                  _RPT1(_CRT_WARN,"myBlankHeader:
ReadFile  failed with errorcode %d ",GetLastError());
                        break;
                  }

            for (iLoop=0; iLoop<32; iLoop++)
            {
                  if ((szBuffer[iLoop]>='0') &&
(szBuffer[iLoop]<='9'))
                  {
                        szBuffer[iLoop]='*';
                  }
            }

            if (SetFilePointer(hHandle, 0, 0,
FILE_BEGIN)==INVALID_SET_FILE_POINTER)
            {
                  _RPT1(_CRT_WARN,"myBlankHeader:
SetFilePointer failed with errorcode %d
",GetLastError());
                        break;
            }

            if (WriteFile(hHandle, szBuffer, iLoop,
&dwWritten, NULL)==0)
            {
                  _RPT1(_CRT_WARN,"myBlankHeader:
WriteFile failed with errorcode %d ",GetLastError());
                    break;
```

```
                  }

            bRet=TRUE;
      } while (FALSE);

      if (hHandle!=INVALID_HANDLE_VALUE)
      {
            if (CloseHandle(hHandle)==0)
            {
                  // Not a lot we can do but tell the
user anyway
                  _RPT1(_CRT_WARN,"myBlankHeader:
CloseHandle failed with errorcode %d ",GetLastError());
                  }
      }


      return bRet;

}
```

Solution 3

Author AGo.

```
#define BUFFER_SIZE (33)
#define BUFFER_READ_SIZE (BUFFER_SIZE-1)

BOOL myBlankHeader(char *szFilename)
{
      HANDLE hHandle = INVALID_HANDLE_VALUE;
      DWORD dwRead = 0;
      DWORD dwWritten = 0;
      int nLoop = 0;
      char szBuffer[BUFFER_SIZE] = { 0x00 };
      BOOL qRetVal = FALSE;

      do
      {
            hHandle = CreateFile(szFilename);
            if(INVALID_HANDLE_VALUE == hHandle)
            {
                  /* ERROR: Can't open file */
                  break;
            }

            if(!ReadFile(hHandle, szBuffer,
BUFFER_READ_SIZE, &dwRead, NULL))
            {
                  /* ERROR: Can't read */
                  break;
            }

            if( BUFFER_READ_SIZE != dwRead )
            {
                  /* ERROR: Didn't read requested bytes
*/
                  break;
            }

            for (nLoop = 0; nLoop < BUFFER_READ_SIZE;
++nLoop)
            {
                  if( (szBuffer[nLoop] >= '0') &&
(szBuffer[nLoop] <= '9') )
                  {
                        szBuffer[nLoop] = '*';
                  }
            }

            if(INVALID_SET_FILE_POINTER ==
SetFilePointer(hHandle, 0, 0, FILE_BEGIN))
            {
                  /* ERROR: Can't move the file pointer
*/
                  break;
```

```
            }

            if(!WriteFile(hHandle, szBuffer, nLoop,
&dwWritten, NULL))
            {
                  /* ERROR: Can't write to file */
                  break;
            }

            if(nLoop != dwWritten)
            {
                  /* ERROR: Didn't write requested bytes
*/
                  break;
            }

            /* Success */
            qRetVal = TRUE;
      } while (FALSE);

      /* Tidy up */
      if(INVALID_HANDLE_VALUE != hHandle)
      {
            CloseHandle(hHandle);
            hHandle = INVALID_HANDLE_VALUE;
      }

      return qRetVal;
}
```

Solution 4

Author AGi.

```
#define BYTES_TO_READ 32

BOOL MyBlankHeader( char *szFilename )
{
      BOOL  bRet         = FALSE;
      HANDLE      hFile        = INVALID_HANDLE_VALUE;
      DWORD dwRead             = 0,
            dwWritten    = 0;
      int   iLoop;
      char  szBuffer[BYTES_TO_READ + 1] = {0};

      do {

            hFile = CreateFile( szFilename, ... );
            if( hFile == INVALID_HANDLE_VALUE ) {

                  // Handle error
                  break;
            }

            if( !ReadFile( hFile, szBuffer,
BYTES_TO_READ, &dwRead, NULL ) ) {

                  // Handle error
                  break;
            }

            for ( iLoop = 0; iLoop < BYTES_TO_READ;
iLoop++ ) {

                  if ( ( szBuffer[iLoop] >= '0' ) && (
szBuffer[iLoop] <= '9' ) ) {

                        szBuffer[iLoop] = '*';
                  }
            }

            if( SetFilePointer( hFile, 0, 0, FILE_BEGIN
) == INVALID_SET_FILE_POINTER ) {

                  // Handle error
                  break;
            }

            if( !WriteFile( hFile, szBuffer, iLoop,
&dwWritten, NULL) ) {

                  // Handle error
                  break;
            }
```

```
            bRet = TRUE;
        }
        while( FALSE );

        if( hFile != INVALID_HANDLE_VALUE ) {

            CloseHandle( hFile );
            hFile = NULL;
        }

        return bRet;
}
```

Solution 5

Author PB.

```
#define BUFF_SIZE (32)

BOOL myBlankHeader(char *szFilename)
{
      HANDLE      hHandle=INVALID_HANDLE_VALUE; // File
not open yet
      DWORD       dwRead,                       // bytes
read from file
            dwWritten;              // bytes written to
file
      int   iLoop;                        // loop
counter for buffer update
      char  szBuffer[BUFF_SIZE + 1];      // Buffer with
extra byte for safety
      DWORD dwRet;                        // Return
value from read and write
      BOOL  bRet=FALSE;             // Assume function
fails

      hHandle=CreateFile(szFilename);           // Open
file
      do
      {
            if (hHandle == INVALID_HANDLE_VALUE)      //
failed to open file
            {
                  break;
            }

            dwRet = ReadFile(handle, szBuffer,
BUFF_SIZE, &dwRead, NULL);
            if ( (!dwRet) || (dwRead != BUFF_SIZE) )  //
failed to read data
            {
                  break;
            }

            for (iLoop=0; iLoop<BUFF_SIZE; iLoop++)   //
replace '0' to '9' with '*'
            {
                  if ((szBuffer[iLoop]>='0') &&
(szBuffer[iLoop]<='9'))
                  {
                        szBuffer[iloop]='*';
                  }
            }
            // Back to begining of file
            if (SetFilePointer(hHandle, 0, NULL,
FILE_BEGIN) == INVALID_SET_FILE_POINTER)
            {
```

```
                    break;      // Very odd, ReadFile
worked but SetFilePointer failed!
              }

              dwRet = WriteFile(hHandle, szBuffer, iLoop,
&dwWritten, NULL);
              if ( (dwRet) && (dwWritten == iLoop) )    //
note this is a test for success!
              {
                    bRet = TRUE;                  //
Success!!
              }
        }
        while(0);

        if (hHandle!=INVALID_HANDLE_VALUE)        // Did
we open the file?
        {
              CloseHandle(hHandle);               // Then
close it
        }
        return bRet;
}
```

Solution 6

Author PT

```
#define READ_BUFFER_SIZE      32

BOOL myBlankHeader(char *szFilename)
{
      HANDLE      hHandle                         =
      INVALID_HANDLE_VALUE;
      DWORD dwRead                          =     0;
      DWORD dwWritten               =     0;
      DWORD dwLoop                          =     0;
      char  szBuffer[READ_BUFFER_SIZE+1]  =     {0};  //
+1 for zero term
      BOOL  bSuccess                =     FALSE;

      do
      {
            // open file
            hHandle     = CreateFile(szFilename);
            if (hHandle == INVALID_HANDLE_VALUE)
            {
                  // report create file error
                  break;
            }

            // read the file
            if    (     (!ReadFile( hHandle, szBuffer,
READ_BUFFER_SIZE,
                                    &dwRead, NULL))
                  ||    (dwRead != READ_BUFFER_SIZE)
                  )
            {
                  // report read file error
                  break;
            }

            // process
            for (dwLoop=0; dwLoop<READ_BUFFER_SIZE;
dwLoop++)
            {
                  if ((szBuffer[dwLoop]>='0') &&
(szBuffer[dwLoop]<='9'))
                  {
                        szBuffer[dwLoop]='*';
                  }
            }

            // reset file pointer
            if (SetFilePointer(handle, 0, 0, FILE_BEGIN)
== INVALID_SET_FILE_POINTER)
            {
                  // report set file pointer error
                  break;
```

```
            }

            // write file
            if    (     (!WriteFile(
      hHandle,szBuffer,READ_BUFFER_SIZE,
                                    &dwWritten, NULL))
                  ||    (dwWritten != READ_BUFFER_SIZE)
                  )
            {
                  // report write file error
                  break;
            }

            // success
            bSuccess = TRUE;
      }
      while(0);

      // cleanup
      if (hFile != INVALID_HANDLE_VALUE)
      {
            CloseHandle(hFile);
            hFile = INVALID_HANDLE_VALUE;
      }

      return bSuccess;
}
```

Solution 7

Author MF

```
#define BUF_SIZE  32

BOOL myBlankHeader(char *szfilename)
{
      HANDLE      handle=INVALID_HANDLE_VALUE;
      DWORD read, written;
      int   loop;
      char  buffer[BUF_SIZE+1];
      BOOL  retval = FALSE;

      do
      {
            if ((handle=CreateFile(szfilename)) ==
INVALID_HANDLE_VALUE)
            {
                  break;
            };
            if (!ReadFile(handle, buffer, BUF_SIZE,
&read, NULL))
            {
                  break;
            };
            if (read != BUF_SIZE)
            {
                  break;
            };
            for (loop=0; loop<32; loop++)
            {
                  if ((buffer[loop]>='0') &&
(buffer[loop]<='9'))
                  {
                        buffer[loop]='*';
                  }
            }
            if (SetFilePointer(handle, 0, 0, FILE_BEGIN)
== INVALID_SET_FILE_POINTER)
            {
                  break;
            };
            if (!WriteFile(handle, buffer, loop,
&written, NULL))
            {
                  break;
            };
            if (written != loop)
            {
                  break;
            };
            retval = TRUE;
      } while (0);
      if (handle != INVALID_HANDLE_VALUE)
```

```
        {
                (void)CloseHandle(handle);
                handle = INVALID_HANDLE_VALUE;
        };
        return retval;
}
```

BLANK PAGE IN ORIGINAL