A MULTI-AGENT ARCHITECTURE FOR KNOWLEDGE SHARING

Sarah Mercer*
H.M.G. Communications Centre
Hanslope Park
Hanslope
Milton Keynes, MK19 7BH, UK
email: sarahm@hmgcc.gov.uk

Sue Greenwood
Intelligent Systems Research Group
Department of Computing
Oxford Brookes University
Oxford, OX33 1HX, UK
email: sgreenwood@brookes.ac.uk

Abstract

A multi-agent system is proposed for knowledge sharing in a system designed to advise on good programming practice. The novelty of the approach is that the agents within the system can learn and adapt domain knowledge to evolve programming standards over time. A prototype system has been implemented which uses the proposed architecture to detect programming defects. The deployment of the system at Her Majesty's Government Communication Centre (HMGCC) is described with test results provided.

1 Introduction

Software development is moving toward a more engineering based approach. Although slow and variable (due to the nature of software), the seeds of good engineering practices are in place within industry. Currently software quality assurance (SQA), structured and formal methods etc, give a framework by which the requirements, analysis and design of a software system can be carried out in a professional manner. Similarly, testing and evaluation of a system is equally well placed. However, basic coding mistakes or lack of good practice can significantly diminish an excellent design. Some of these issues can be aided by software metric style evaluation but they are not all encompassing.

The nature of programming is still, in most practical scenarios, something of an art form. The plethora of languages, constructs and ways of tackling the same problem make this a hard area to apply engineering principles to, without being prescriptive. Note that the reasonable assumption is that the design under discussion does not arrive on the programmer's desk with complete line-by-line pseudo code.

In the real world, especially on rapid turnaround jobs, coding remains the main creative process within an organisation. Experienced programmers are always respected and have proved invaluable in reviews to find problems that did not arise at design review and are not identified by normal means (such as the compiler). This experience leads to the concept of 'Best Practice' where traps, pitfalls and maintenance issues are handed down or presented in document form.

Due to the obscure and indefinite nature of this problem, it would appear that it is not feasible for a standard system to provide a solution; there is a need to introduce a degree of intelligence into a system in order to provide practical application of the theory. As mentioned, some coding errors can be detected and rectified by the use of existing SQA methodologies: such as guidelines for good design and implementation. 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 [Pressman, 1992]. However, such formal and mathematical approaches are not flexible or powerful enough to replace the human "gut-instinct" that is based on experience.

1.1 System aims (the problem)

The system is designed to aid the engineer in the following areas: coding guidelines, good design, program correctness and implied knowledge. These are detailed below:

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 only those rules that are undisputed and easy to enforce.

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 fair amount of design is still left to the programmer (this assumes the real world situation of a 'thin' design specification arriving on the programmers desk). 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.

* This research is sponsored by Her Majesty's Government Communications Centre.

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.

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, which 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.

2 Related Work

There are tools available that statically analyse source code, for example; Nottingham University's Ceilidh system [Zin and Foxley], Ayse Salmon's teaching assistant [Salmon, 2001], LCLint [Evans], and NuMega's Code-Review [Numega].

The Ceilidh System marks students coursework for a programming module. It uses many of the previously mentioned metrics to measure correctness, maintainability and efficiency. Although it is very good at measuring the quality of the coursework, it is unable to make suggestions to the student to aid the students learning.

Ayse Salmon's teaching assistant focuses on the factors of correctness (dynamic) and maintainability. It is able to mark the students work, but is also unable to give advice. Both of these systems procedurally review the code and then use metrics to gain a measure of the quality but take the process no further.

LCLint is a system that has been used historically in the HMGCC programming environment for checking source code. It acts as an advanced compiler preprocessor. This system does give advice about the problems found in the code, but it is very tightly linked to the language is it is reviewing, and is unable to measure general qualities such as complexity.

CodeReview from NuMega is a commercial system that reviews Visual Basic (VB) source code. This system gives feedback using the VB help environment by directing the user to appropriate pages as solutions. CodeReview allows a standard to be set centrally for all members of a team of programmers, to ensure quality is consistent.

None of these systems are able to adapt to the style or level of competency of the programmer. Neither can these systems learn from previous experience or outside influences. All of the systems assume that the set of coding guidelines will be static over time and none of them directly use external influences (peer-reviews, experienced engineers) to evolve their standards.

3 Proposed solution

3.1 System Requirements (refined aims)

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 given a number of reasons (not least it is not technically viable), but also the knowledge within the system must allow for change. This reflects the way organisation's guidelines evolve over time, taking into account new environments, influences and ideas, and changes in the level of quality required. 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 will 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 [Pomberger, 1984].

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 [Perrolle, 1995], to individual users especially newcomers.

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 *off-line system* is best suited to their environment, i.e. it should wait until prompted before giving its opinion and not annoy like the Microsoft 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, the errors removed and warnings minimised.

Usage

Establishments that use coding guidelines usually enforce them by a formal end-of-stage code review and sometimes at milestone points, conducted during the

implementation stage of the software lifecycle. The most common format is the peer-review, where a team of engineers review the code individually, and then meet to form a consensus of findings. This allows not only the quality of the code to be measured and improved, but also allows for the process itself to evolve. This is where new guidelines are found and better solutions are provided, which is imperative due to the obscure nature of software quality assurance. It is interesting to note that at this stage 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 3-5 people (5 or 6 being the upper limit for a piece of code) the number of anomalies pinpointed is significantly greater than a review with just 1 or 2 engineers.

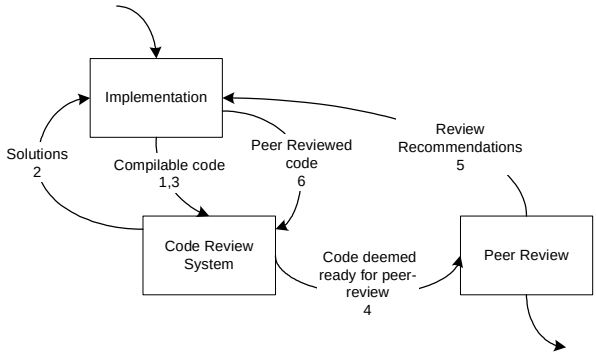

Figure 1: Life cycle including Code Review System.

Figure 1 shows the way in which a code review system could be used to minimise the number of defects found in code, prior to peer review.

We envisage a system, which comprises a set of learning systems, that adapt to their users and represent the user's practices and preferences in a team environment. It is hoped that this system will exhibit similar strengths to the peer-review format; 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.

3.2 System Design

The system comprises multiple nodes. Each node is a standalone system that works for and represents an engineer.

Source code is read into the system, checked that it is syntactically correct and translated into an appropriate representation. This representation is then used to extract basic information about the code, to give a framework of the code, upon which defect detection can be facilitated. Interrogating this extracted information and comparing it against an initial set of guidelines enables anomalies in the code to be detected. Solutions can then be suggested as workarounds or fixes for the identified anomalies, these solutions are provided from the system's local knowledge, or by asking other nodes for possible solutions.

Each node is able to learn new guidelines and solutions, and is able to learn both by direct feedback from the user and by identifying repetitive behaviour that conforms to the guidelines and applying modified solutions to similar situations (case-based reasoning). As nodes share solutions and guidelines, and each node records information about the engineer supplying them, an internal view of the team is developed. Over time, agents that represent experienced, or reliable, engineers are recognised and so too are learners. This has a bearing on suggestions made to those users and to the weighting of confidence in suggestions given. An initial view of the team is given using stereotypes. Learning reinforcement and knowledge refinement will be derived from the feedback of results of the formal peer-review. The outcome of which will have precedence over initial states.

A global view of the team and a central set of guidelines are also gathered by monitoring all interactions between nodes. The central set of guidelines can then be reviewed to check that the consensus of the system matches that of the team.

3.3 Why Agents

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. To this end, the nodes will need to be autonomous in the attainment of their goals, such that their use of speech act base communication and knowledge enables them to co-operate to intelligently derive, from information gathered from a set of individuals, a reinforced and agreed consensus. These requirements echo the main aspects of agency, specifically the need for autonomy and co-operation, it is assumed that this will afford the system the same benefits as the peer-review, where the whole is worth ore than the sum of its parts.

3.4 Agent-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-operating within each node. The high-level agent architecture is shown in Figure 2.

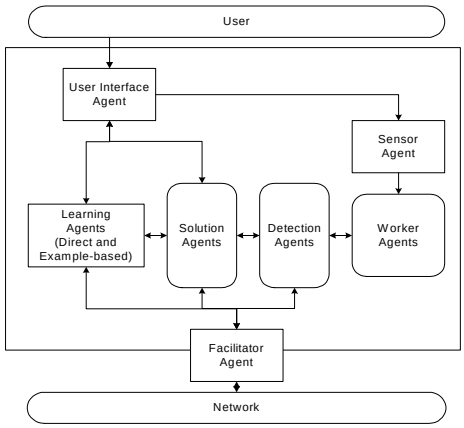

Figure 2: Proposed agent architecture.

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 codes 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 [Finin, 2001]. The BDI model allows practical reasoning, which is reasoning directed towards actions [Wooldridge and Parsons, 2001], by analysing the current environment, its own knowledge (the beliefs) and goals (or desires) the agent can decide what to do next, and what it needs to do to achieve it (its intentions or plan). 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 and therefore achieve their goals.

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), [Kolodner, 1993; Leake, 1996], 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 the systems knowledge.

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

As shown in Figure 1, when the user has made the appropriate alterations, the code is entered into the system again, where the learning agents can adapt their knowledge based on solutions that have been implemented and upheld by the user and 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-review. The results of which allow the system to refine and reinforce its knowledge. The revised code that includes the peer-review recommendations is again entered into the system, where the system can again adapt based on, where new defects have been spotted, resulting in a new or amended guideline; where a known defect has been ignored, resulting in an amended guideline; where suggestions have been overturned, resulting in a solution being modified or replaced.

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 evolution of the system, as negative or not preferred solutions will *die out* over time.

By monitoring the user selection and the outcome of peer-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 the user's 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 teams current level of quality.

4 The Prototype System

A subset of the system has been implemented as a prototype and tested in the real world situation of HMGCC. The aim of the prototype is to assess the knowledge representation and the ontology used, for (non-learning related) reasoning, therefore ensuring that the next stage of implementation (learning) has a stable foundation upon which to work. Certain assumptions have been made; the preferred software language, and therefore the language to be reviewed, at HMGCC is 'C' as defined in the MSDN[1], the system is limited to only detecting defects from the initial set.

4.1 Implementation

The prototype was implemented in Java, using JESS (Java Expert System Shell) as the agents' internal reasoning engine (IRE) [Friedman-Hill]. Java was chosen as it is the most appropriate language for this type of rapid development and integration with Jess. Figure 3 shows the subset of the main architecture that has been implemented as the stage 1 prototype.

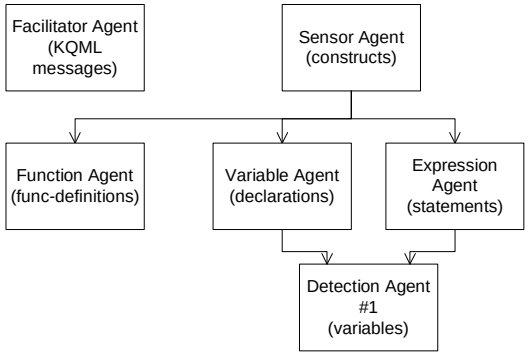

Figure 3: Prototype design.

[1] Microsoft Developer Network, Visual C++, C Reference.

Knowledge & the Domain

Representations (syntax trees) are shown for the statement “a=a+1;” and the declaration “int a, b=4;”

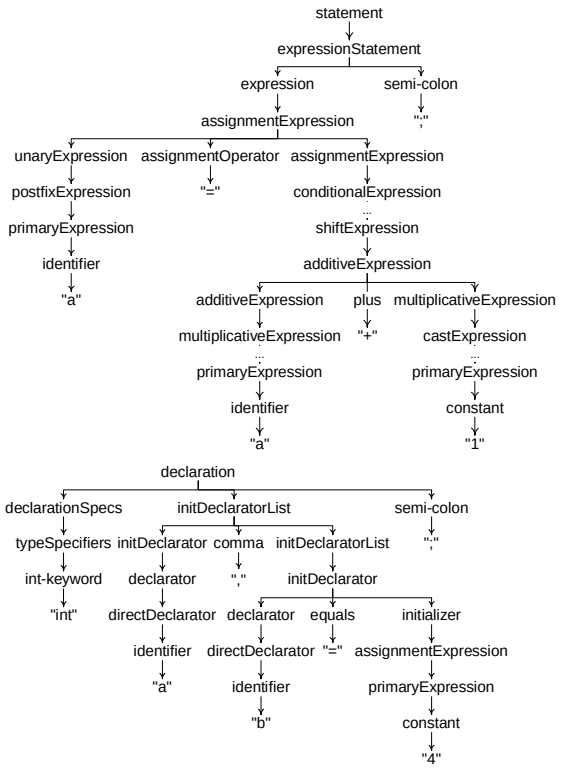

Figure 4: Syntax trees, as constructed by Sensor Agent.

The representations shown in Figure 4, allow the agents to interpret the meaning of the constructs, for example given 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, given the second tree, the leaf nodes of branches which contain “directDeclarator” are the names of the variables declared.

Communication

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

The syntax used both internally and shared between the agents is JESS, which is loosely based on that used by CLIPS and is highly expressive. This system allows full interaction between the rule base system and Java objects, giving powerful flexibility.

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

The pragmatics are based on KQML [Finin *et al*, 1993], which describes the way the agents communicate, i.e. how to ask a question, how to respond to a question, etc. The current set of performatives used is; ask, tell, reply, subscribe, advertise and recommend.

Ontology

Each agent contains a JESS expert system, as its IRE. Each agent’s IRE is initialised with the ontology of the system. The main items of knowledge for the worker agents are the syntax trees that represent the constructs, as previously described. Due to the complete nature of the source code domain, parts of the ontology can be expressed in terms of tree parsing functions.

Sensor Agent

There are no goals defined within the Sensor Agent, as it is purely reactive. When instructed to by the user (via the interface) it reads in the source code and produces a representation for each construct contained within. It is advertised as the “expert-in” constructs, streaming the set of constructs on demand to other agents.

To produce a representation of the source code, the Sensor Agent has a second JESS rule base that is initialised with C specific knowledge. The sensor agent is the only agent that includes language dependent knowledge. This clear separation of the C specific knowledge and the agents IRE allows the supported language to be replaced easily. One obvious advantage of working in the knowledge domain of a programming language is that it already has a complete and correct specification that can be easily ported to a set of rules ready for inclusion into the system.

Expression/Variable/Function Agents

These agents are very similar in their aims. They receive constructs that represent statements, declarations and function definitions respectively. A small amount of reasoning is used to ascertain rudimentary information that is available to other agents on demand.

Detection Agent #1

The goal of the detection agent is to detect defects in the code that are related to variable usage. To achieve this the set of variable declarations are requested from the variable agent, for each of these variables a request is made to the expression agent to ascertain how the variable is used and manipulated during the flow of the program, a conclusion can then be drawn about the appropriate use of the variable, etc.

4.2 Testing

The test plan aimed to prove the legitimacy of the architecture by ensuring that the first stage of implementation provided a good foundation, upon which the learning mechanism could be implemented. To ensure this the system was compared to others, in its ability to parse the source code, produce the correct syntax trees and detect defects in the code, from an initial set of guidelines.

For the 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 reasoned over the elements within the code.

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

The test results showed that 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 recognise variables that should be made constants. This highlights that the prototype system is able to detect anomalies in the code, which are not just related to correctness, as recommending a variable be made constant is a typical peer-review recommendation to improve the maintainability of the code.

We believe that it is evident that the prototype is able to compete with static analysis tools in the area of correctness, and out perform them in other areas of software quality such as maintainability. Therefore we also feel, that this is an appropriate foundation, upon which to proceed with implementing the full architecture.

5 Discussion & Conclusion

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 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.

In the homogeneous architecture of the prototype, a strong development benefit emerged in respect to 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.

Further work will include the implementation of solution and learning agents that will facilitate case-based reasoning, such that the system can learn both new defects and solutions. The next stage of the research will be concerned with the feasibility of the BDI architecture in allowing the detection agents to co-ordinate and share information to achieve their joint goal of ensuring all guidelines are being adhered to, and to investigate whether this architecture is appropriate for implied correctness guidelines.

References

[Pressman, 1992] Roger S. Pressman. *Software Engineering: A Practitioners Approach*. McGraw-Hill, 1992.

[Zin and Foxley] Abdullah M. Zin and Eric Foxley. *Automatic Program Assessment System* http://www.cs.nott.ac.uk/~ceilidh/papers/ASQA.html

[Salmon 2001] Ayse Salmon. *Unpublished PhD Thesis*. OBU, 2001.

[Evans] David Evans, project leader. *'LCLint/Splint Homepage'*. University of Virginia, http://lclint.cs.virginia.edu/.

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

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

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

[Finin, 2001] Tim Finin. Tutorial on Agent Communication Language, *Autonomous Agents 2001*, Montreal, Canada, May 2001.

[Wooldridge and Parsons, 2001] Michael Wooldridge and Simon Parsons. Tutorial on Rational Action in Autonomous Agents, *Autonomous Agents 2001*, Montreal, Canada, May 2001.

[Kolodner, 1993] Janet Kolodner. *Case-based reasoning*. Morgan, Kaufmann, 1993,

[Leake, 1996] D. B. Leake. *Case-based reasoning: Experiences, Lessons & Future Directions*. AAAI, 1996,

[Friedman-Hill] Ernest J. Friedman-Hill. *JESS Homepage*. http://herzberg.ca.sandia.gov/jess/

[Finin *et al.*, 1993] Tim Finin, co-chair. *Specification of the KQML Agent-Communication Language*. http://www.cs.umbc.edu/kqml/kqmlspec/spec.html.