Legalising reliable computer programs*

Paul Mc Kevitt
Department of Computer Science
University of Exeter
Exeter EX4 4PT, United Kingdom
E-mail: JANET: pmc@uk.ac.exeter.cs or pmc@cs.exeter.ac.uk
(0392) 264061

Abstract

One of the most serious problems with computers today is that they are not reliable. The reason for this is quite simple. Computer programs are built to solve particular problems and these problems can be quite large. By large we mean having lots of constraints, cases and rules as to what constitutes the problem. Real world problems are so complex that it's hard to be sure that when a program has been built it will never fail for an input that has not been taken into account. Reliability of computer programs has serious legal implications because if a program fails to work then a question arises as to who is responsible. If the field of computer science concentrates on better techniques for increasing computer reliability then we will decrease questions of responsibility. It is argued that the root of the problem of responsibility is the problem of reliability and that computer scientists should be worrying about the software reliability problem.

*This research has been funded in part by **U S** *WEST* **Advanced Technologies**, Denver, Colorado, under their Sponsored Research Program.

1 Introduction

One of the largest problems with computers today is that they are not reliable[1]. It is very difficult to build a computer program which will never fail[2] as there may be certain inputs not taken into account. The problem of unreliability in programs is one of the causes of legal debate. The further problem of responsibility arises. Who is responsible for the failure of the program? Is it the programmer, or the employee who recommended the program for the company, or the systems person who placed it on the computer? The point is put nicely by Wilks and Ballim (1991, p. 119), as they speak of responsibility: "The difficulty [of assigning punishment to programs themselves] can be avoided by always identifying humans, standing behind the machines and programs as it were, to carry the blame, in the sense in which there are always real humans standing behind agents, and behind companies, which also have the legal status of nonhuman responsible entities ("anonymous persons" in much European law)." It is argued here that the problem should be tackled at the source, and that more effort should be placed on program reliability. More reliable programs will reduce questions of responsibility.

2 Existing reliability techniques

Computer science has already developed techniques for ensuring the reliability of computer programs. Programs have been developed to aid programmers in developing and testing software for specific domains. Practical software engineering tools such as CASE (Computer Assisted Software Engineering) products are being used today (see The Byte Staff 1989). CASE tools are very useful for building computer software for limited domains, yet not very useful for tackling problems outside their scope. For example, there are very few CASE tools which would be useful for tackling the problem of building a machine translation system for translating Swahili into Chinese.

[1] Any references to reliability in this paper will be to software reliability although the problem of hardware reliability is also important.

[2] By "fail" we mean that either (1) the program crashes, in the common computer science sense of crash, or (2) the program has no procedures for handling the input in question.

Methodologies for the development of computer software have also been defined. Partridge (1986) argues for the a Run-Debug-Edit methodology which is later modified to RUDE (Run-Understand-Debug-Edit) in Partridge and Wilks (1987). RUDE is a software development methodology for the design of, mainly, AI programs. It is also pointed out by Partridge and Wilks (1987) that the RUDE methodology may be useful for traditional computer science problems too. This methodology calls for a discipline of incremental program development where programs are run and, if they fail on input, are edited and rerun. The problem with RUDE is that it is tedious, and takes a long time, as the programmer is just hacking piecemeal at solving the problem without really knowing what the problem is. There is no specific goal toward which the program or programmer is geared. Also, Partridge[3] has pointed out that in many cases it is difficult to solve an AI problem, piece by piece, because solutions for decomposed parts of the problem do not solve the problem as a whole, when placed together.

Another methodology called SAV (Specify-And-Verify), coined in Partridge (1986), calls for proofs of specifications of problems, and formal verification of the subsequent algorithm. The SAV approach is advocated by by Gries (1981), Dijkstra (1972), and Hoare (1981). The use of formal techniques in proving programs correct for real world complex problems in computer science has proven difficult. One of the problems with Artificial Intelligence (AI) programs is that they are very difficult to specify (see Partridge 1986). The application of proof logics to the intricacies of complex programs is too tedious. The technique has only become useful for small, simple programs.

Both RUDE and SAV only ensure that a program is reliable for a particular specification. There is no guarantee that the specification is correct, or reliable, for the real world problem at hand for which it is intended. Also, taking into account the fact that AI programs are very difficult to specify these will have more serious implications for computer reliability than traditional computer science programs. New techniques are needed for reliability checking for traditional computer science and AI problems which may incorporate elements of both RUDE and SAV.

---

[3]Presentation at the Fifth Rocky Mountain Conference on Artificial Intelligence (Theme: Pragmatics in Artificial Intelligence), Las Cruces, New Mexico, June, 1990.

3 The EDIT methodology

EDIT (Experiment-Design-Implement-Test) first published in Mc Kevitt and Partridge (1991) is a software development methodology which attempts to integrate elements of the SAV and RUDE methodologies. EDIT incorporates experimentation as an integral component. This is particularly useful in the AI problem domain which incorporates the added difficulty of the researcher not knowing how to describe a problem while trying to solve it. EDIT is both a (1) software development, and (2) software test methodology. EDIT addresses the problem brought out by Narayanan (1986, p. 44) where he says, "The aim of this paper, apart from trying to steer well clear of terminological issues, such as the distinction between 'science' and 'study', is to demonstrate that unless AI is provided with a proper theoretical basis and an appropriate methodology, one can say just about anything one wants to about intelligence and not be contradicted; unless AI is provided with some reasonable goals and objectives little of current AI research can be said to be progressing." It is believed that EDIT might be the methodology that Narayanan asks for.

Briefly, EDIT has the following stages:

1. **Experiment:** Experiment(s) (E) are conducted to collect empirical data on the problem. This data can be stored in log file(s) (L).

2. **Design:** A design[4] (D), or specification, can be developed from L together with relevant theories of the program domain.

3. **Implement:** The description, or specification, is implemented (I) as a computer program (P).

4. **Test:** P is sent around the cycle and tested by placing it through E again. However, this time E involves P whereas initially E did not involve a program. The cycle is iterated until a satisfactory P is found.

The system developer(s) initially use(s) E to help define the problem, and successively use(s) E to develop and test P. In the initial stage E does not involve a program for responding to questions. However, each subsequent E involves a partially implemented P, until the final P is decided upon. The

[4] By "design" we mean any reasonable description whether it be in English, Hindi, Gaelic, logic, algorithmic form, or assembly code.

design of the initial E is up to the AI theorist who wants to build a program P. Of course, this design will limit the scope with which the program can be tested. The design of the initial P, and each subsequent P, is also up to the AI theorist. EDIT will always terminate after E and before I in the cycle. The EDIT cycle is shown in Figure 1 below.

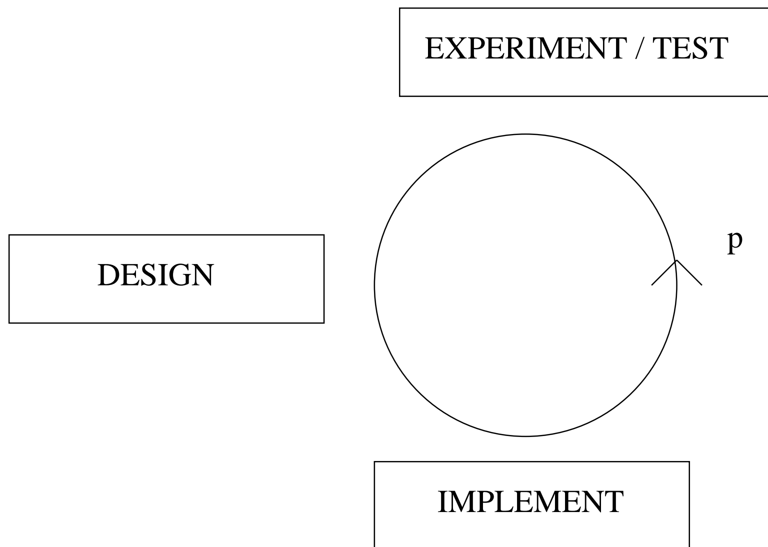

Figure 1: EDIT (Experiment Design Implement Test) Cycle.

At the experiment stage (E) an experiment is conducted to gather data on the problem. Say, for example, the problem is to develop a natural language program which answers questions about computer operating systems like UNIX. Then, valid experimentation software would be a program which enables a number of subjects (S) to ask questions about UNIX, and an expert to answer these questions. An example setup for this experiment would be the Wizard-of-Oz[5] paradigm. A number of S and Wizard's (W) from varying backgrounds may be used in the experiment. Of course, the greater the number of S and W the more comprehensive the data collected will be.

[5]A Wizard-of-Oz experiment is one where subjects interact with a computer through typed dialogue, at a monitor, and are led to believe that they are conversing with the computer. For example, in the case of a Wizard-of-Oz test for a natural language interface, a subject's utterances are sent to another monitor where a "Wizard", or expert, sends back a reply to the subject monitor.

Also, there may be groups of S and W rather than just a single S and single W. Information on exchanges between S and W is logged in a log file for later inspection. S and W operations are marked in the file. Such an experiment is described with greater detail in Mc Kevitt and Ogden (1989) and the implications of that experiment are described in Mc Kevitt (1990b).

At the design stage (D), L from E is analysed and inspected. In the initial stage the data here gives a snapshot description of the problem and how it is characterised. Further stages of the cycle will give snapshots of how well the problem is characterised in the current P. An analysis of L will give a picture of the information needed in various components of a program, such as knowledge representation, user modeling, and reasoning components. Many researchers and domain experts from various backgrounds may be called in to analyse L and determine what aspects of the software need to be developed. In fact, the type of researcher brought in will determine the type of program eventually developed and the best of all worlds would be to have a wide span of researchers/experts from different backgrounds. The job of the researchers is to develop algorithms with the help of the data, and to specify these algorithms in some manner.

At the implementation (I) stage the algorithms or designs in D are implemented. These designs may be implemented in any programming language (P) that the implementers find most appropriate. Finally, the implemented program is sent back to E again and tested. Then, a new cycle begins.

During the initial state of the EDIT manifestation described here, S and W interact over the problem and the data is logged in L. Data may be collected for a specific task within a domain, or the whole domain itself. Each successive run of E involves the incorporation of P, which tries to answer questions first and if it fails W steps in while P restarts. The cycle may be operated in real time, or batch mode. In batch mode the experimentation component would involve a number of batched questions which are collected from S, and processed by P, where W interrupts if P fails.

The EDIT cycle continues until the program performs satisfactorily to the requirements of the designers. The deigner(s) may wish P to perform satisfactorily only 50% of the time, or 80% of the time. The success or failure of P will be determined at design time, D, when the L is analysed. L will show where P has failed and where it has passed the test. W entries will show up why P did not work and will indicate what components of P need to be updated. In the case of natural language question answering W entries

might show up the fact that certain types of question are not being answered very well, or at all. Therefore, W entries would indicate how P needs to be augmented in principle to solve a recurring pattern of failure. W entries could be analysed for such recurring patterns. In effect, what is happening here is that P is "learning" by being investigated, and augmented, in the same way as a mother might teach her child, noticing the child's failure to complete certain tasks[6].

The success of P is measured by the number of answers P can give, and the number of answers P gets correct. The measure of capability and correctness is determined by inspection of L. During the development of P the initial coding may need to be recoded is some manner as data collected later may affect P's design.

There are many forms in which the EDIT cycle may be manifested. The experimentation stage may involve experiments other than the Wizard-of-Oz type. Another experiment might involve an observer sitting beside the subject during testing and helping the subject with the program as he/she uses it. The observer would restart the program if it failed. Such a technique is used in experiments described by Al-Kadurie and Morgan (1989). The design stage and inspection of L may involve only one or a number of designers. These designers may know much about the domain, and little about design, or vice versa. Experts and good designers may both be used at the design stage. Also, E could consist of a set of experts with different points of view and different backgrounds. Bear in mind too that the interface to P need not be a natural language one. We have chosen natural language because it seems to be the ultimate interface, but we could well have chosen a more limited menu-based interface with icons, where the interaction could easily have been logged. Also, it could be argued that a natural language interface is one of the most general possible, and that other more limited interfaces are subsets, or equivalent, in terms of power to it. Hence, EDIT may consist of many manifestations of the methodology, yet, the basic methodology involves developing and testing through experimentation.

EDIT can be used to develop and test systems. Development continues until the designers are happy with the system at the completion of some cycle. EDIT can also act as methodology for testing hypotheses in AI where

[6] This analogy was provided by personal communication from Brendan Nolan of University College Dublin.

such hypotheses may be solutions to parts of problems. The advantage of the Wizard-of-Oz technique incorporating W is that if P fails for reasons other than the hypothesis then W can step in, keeping P alive, so to speak. Meanwhile, no data is lost in the current experiment dialogue. Of course, the log file marks where W interrupts. During testing as far as S is concerned the program has never failed as S does not necessarily know that W has intervened. The data from the testing phase can be logged in a file and system developers can then observe where the system failed, and where W interfered. This information will be used in updating the system, and any theory which the system represents.

EDIT is a useful technique in that it allows the iterative development of systems and gives feedback on how to design an AI system as it is being developed. Sharkey and Brown (1986, p. 282) point out that the belief that an AI system can be constructed first, and then tested later, as argued by Miller (1978), is not the way to go. Sharkey and Brown show that (1) an AI system takes a long time to build, and it may be wrong at the beginning, and (2) an AI theory, and its implementation in the final state, may not be coined in a way that allows psychological testing.

One of the problems with AI today is that it is not appreciated as a science and has no scientific test methodology. Narayanan (1986, p. 46-47) points out: "It can be argued that the criterion of implementability is vacuous at the level of the Church-Turing thesis." The thesis basically says that any process which can be described by an algorithm can be implemented on a computer. Thus, any AI theory which can be described by an algorithm can be implemented on a computer, and hence all AI theories are valid no matter what they say. Sharkey and Brown (1986, p. 278) also point out this problem: "To say that a theory is implementable is simply to say that it can be expressed in the form of a computer program which will run successfully", and suggest that a solution needs to be found: (p. 280) "Another question we would like to raise here is this: At what point in implementation do we decide that there are too many patches to accept that the running program is actually a test of a theory." Sutcliffe (1991) argues for more empiricism and says, "I see the use of norming studies and other techniques from psychology as being relevant to AI." EDIT calls for not just implementability but also for the implementation to work on experimentation over real data. Also, EDIT solves the problem of how to check whether an AI theory is valid. Narayanan (1986, p. 48) points out: "In any case, even if a criterion of complexity for

AI programs (theories) can be found, there still remains the suspicion that no criterion exists for determining whether an AI theory is true or accurate." EDIT may provide such a criterion in the inspection of log files.

To those researchers who might argue that EDIT may be narrow and inductivist, as an hypothesis test method we would point out that EDIT is totally compatible with a test, where an initial design has been thought up, and programmed, and where P is entered into EDIT, at E, before any data is collected. Hence, EDIT is completely in accord with scientific test methodologies proposed by the philosophers of science, Hempel (1966), and Popper (1972). EDIT is a methodology for testing hypotheses in AI where AI is a scientific endeavour. EDIT does not follow the traditional AI approach of building programs which are stated to embody a theory, and where implementability is the test of this theory.

4 Comparing EDIT to RUDE

EDIT is not just a rearrangement and renaming of RUDE. The difference is that EDIT offers a means of convergence on a solution. EDIT and RUDE differ in that the algorithms are developed in conjunction with data describing the problem rather than from what the problem "might" be. Too often in the field of AI there are attempts at, a priori, deciding what a problem is without any attempt to analyse the problem in depth. As was pointed out earlier one of the problems with developing AI programs is that it is very difficult to specify the problem. One solution to that might be to collect data on the problem, as EDIT calls for. The second major difference is that experimentation involves testing software over real data in the domain. Also, by using the Wizard-of-Oz technique the testing phase breaks down less as the wizard keeps the system going. We argue that this is important because if a test fails then data can be lost due to temporal continuity effects. Failure happens a lot while testing AI programs. For example, if one is testing a natural language interface, with an hypothesis for solving reference in natural language dialogue, then if the test fails the continuation of that dialogue may never happen, and data will be lost.

The problem with RUDE is that it does not include any goal as part of the process of development; only the update of a program. We argue here that E must be included to produce log files which measure how close P is

to the goal that needs to be achieved. EDIT can be considered a more "tied down" version of RUDE where it is clearer what the problem is, and how well P is solving the problem. In fact Partridge and Wilks (1987, p. 117) say, "What is needed are proper foundations for RUDE, and not a drift towards a neighbouring paradigm."

EDIT is an attempt to address the problem brought forth by Narayanan (1990, p. 179) where he says: "What we need here is a clear categorization of which edits lead to 'theory edits', as opposed to being program edits only. It is currently not clear, in the AI literature, how such a categorization might be achieved. AI does not have the sort of complexity measure which would help identify when the theory, as opposed to the program, should be jettisoned in favour of another theory." Using EDIT an inspection of L should show up, in many cases, where a program has failed because of an hypothesis failure, or because of other reasons, and hence there will be distinct implications for the theory and the program. Also, Narayanan (1990, p. 181) says: "But given the above comments, it appears that there can, currently at least, be no scientific claims for claiming that one AI theory is better than another and that AI is making progress, simply because the conceptual tools for measuring one theory against another, and so for measuring the progress of AI are missing." We believe that EDIT may be a path on the road to such conceptual tools. It may be the case that EDIT has a lot to say in the development of foundations for AI as a science rather than a technology (see Narayanan 1990, Partridge and Wilks 1990).

The EDIT cycle is conducted until the implementation performs satisfactory over a number of tests. The EDIT cycle enables the iterative development of a system through using the problem description itself as part of the solution process. EDIT is not just an hypothesis test method, but is also a method by which the *reason* for failure of software is logged and a method where that reason does not cause data loss. EDIT is useful for the development of software in an evolutionary way and is similar to those techniques described in Connell et al. (1989). Again, 100% reliability is very difficult to guarantee but we believe that problem description and implementation through experimentation will lead to better implementations than RUDE on its own.

EDIT is like the general methodology schemes proposed by researchers who are developing expert systems. The stages for the proper evolution of an expert system are described by Hayes-Roth et al. (1983):

- IDENTIFICATION: determining problem characteristics
- CONCEPTUALIZATION: finding concepts to represent knowledge
- FORMALIZATION: designing structures to organize knowledge
- IMPLEMENTATION: formulating rules that embody knowledge
- TESTING: validating rules that embody knowledge

This is in the spirit of EDIT where, of course, identification is similar to E, conceptualization and formalization to D, and implementation to I. However, with EDIT, E is involved in both identification and testing and we argue that this is the way to go about testing if P is to meet the problem head on.

EDIT is currently being used in the development of AI software which answers natural language questions about computer operating systems. An initial computer program was developed called OSCON (see Mc Kevitt 1986, Mc Kevitt 1990a, Mc Kevitt and Wilks 1987, and Mc Kevitt and Pan 1989) which answers simple English questions about computer operating systems. To enhance this research it was decided that an experiment should be conducted to discover the types of queries that people actually ask. An experiment has been conducted to acquire data on the problem. More details on the experiment and its implications are given in Mc Kevitt (1990b).

5 Applying EDIT to the law domain

EDIT can be applied in the law domain as follows. One possible manifestation of the experimentation stage (E) would be where a Wizard-of-Oz exeriment has been set up such that W is a lawyer or solicitor giving advice or answering questions. A number of subjects, or clients, sit sown and ask questions on law and W answers these constituting a sort of lawyer-client interaction. In effect, we have what Mehl (1959) called "The Law Machine" which would "provide a decision within a highly specialised field of law ... [or, more ambitiously] ... answer any question put to it over a vast field of law." The program P would act as a solicitor, or lawyer, advising S, or the client, on solutions to problems.

The domain of law may be fixed to domains like *Law relating to Supplementary Benefits and Family Income Supplements* ("the Yellow Book") (see Bench-Capon 1987) which incorporates acts of parliament and statutory instruments — a domain used by Kowalski and Sergot (1991). Indeed, Kowalski and Sergot (1991, p. 105) point out that, "to construct a program dealing with the British Nationality Act that could be used in practice, it would be necessary to enlist the help of an expert lawyer, preferably several experts on citizenship and immigration law."

In another light, the program P may be intended to be a judge, where P is fed with evidence, and cases for and against the defendant. Here, E would involve both members of the prosecution and defence interacting with the W which may be a judge or set of judges. In effect, P would be simulating a courtroom scene. This would involve a very complex experimental setup indeed. Yet, other sciences have such complex experiments. This scenario is pointed out by Bennun (1991, p. 48), as he says: "The problem is not to computerize the *law* (his italics), which is clearly feasible through the use of logic and databases; it is to computerize the judicial *method* (his italics), which is another matter entirely."

The design phase (D) would be one where a number of solicitors and programmers perused the log files and determined the right information to be placed in the P. The information could be obtained from legal texts and experts. In the case of P acting as a judge information would have to be obtained on the running of the judicial process itself. This process of entering information into such a program is termed "legal knowledge engineering" by Clark and Economides (1991, p. 7), "The process of "extracting" legal knowledge from legal texts and legal experts and representing it within an expert system is referred to as "legal knowledge engineering."

The design would be implemented at stage (I) and the program retested over S. The cycle is completed when the required measure of success is reached.

In the above discussion P is described as a program which either acts as a legal adviser, or the judicial process itself. P may also assume any of the types of legal work as defined by Clark and Economides (1991). These are (a) giving legal advice, (b) negotiating points of view of other lawyers (different W's would be used here), (c) officials and parties representing clients in court, and (d) structuring of the law itself. The program P may be build to cater for these more specific tasks.

One of the problems arising in the law domain, and other domains, would be to determine the W to be allowed to answer questions by S. The problem with this is that, say, W on the whole were logical positivists (see Clark and Economides, 1991 p. 7), then P might turn out to have a logical positive point of view. On the other hand if they were naturalists then we may have the opposite effect. Where is the line drawn? Also, the inspectors of log files during D may be partial in their analysis of the L and even the programmers during I may be partial too. In fact, subjectivity is a problem with most software development today. The solution to this problem would be to organise a balanced set of W, inspectors and programmers who represent a balanced set of views. In effect, this is argued for by Whitby (1991, p. 97): "A useful step might be the introduction of something along the lines of an 'ethical committee' which could consider some of the legal, social, and political implications of AI systems in law which present special problems. The criteria for membership of such a committee are by no means clear, but almost any committee would be more effective and carry more consensus than the present reliance on ad-hoc decisions, often by researchers with limited legal knowledge, and without public scrutiny or debate." In EDIT such debate and scrutiny might come in during any of E, D, or I, and is most important during D.

The EDIT methodology does not follow the spirit of Kowalski and Sergot (1991, p. 101) in which they argue for a set of representations where "The most obvious application of the British Nationality Act program, for example, is to test whether a given individual is, or is not, a British citizen according to (a given interpretation of) the Act. There is no fundamental reason, however, why the same representations should not be used for other legal problem-solving tasks: in systems which plan or advise on a sequence of legal transactions to achieve some desired goal, or to help identify for a lawyer possible lines of reasoning and argumentation, or in systems which are intended to help in the formulation of legislation itself." We believe that a set of representations may be determined a priori but that those representations will change as more data is gathered during the EDIT cycle. In fact, a Wizard-of-Oz experiment on a natural language help system for UNIX showed up a number of problems with the design of existing such help systems (see Mc Kevitt 1990b).

Also, Kowalski and Sergot (1991, p. 108) point out: "One can imagine how a draftsman who is charged with formulating a piece of new legislation or

with modifying an existing piece might use an executable model of his current draft to test that it "does" what he intended. He could have available, for example, a library of stereotypical cases on which he can try out changes to his current draft." We argue that the EDIT approach is a better test as it involves testing P as it should behave in actual use.

The development of an EDIT approach is, in fact, argued for by Clark and Economides (1991, p. 22), where they say: "However, if we are to design computer systems that are capable of supporting processes of intelligent human interaction and communication in legal settings, we need a far more detailed understanding of these processes than we have at present" It is argued here that such processes are given by the log files. Also, Jackson (1988, p. 115) says: "We need empirical studies which will trace, step by step, the forms of discourse by which legal messages are constructed and communicated. Each of these steps in the communicational chain requires independent assessment; who is communicating to whom?, in what medium?, and through the use of what codes?"

The distinction between the use of EDIT as (1) a development methodology, and (2) a test methodology is brought out by Susskind (1991) in his discussion of the distinction between pragmatism and purism. EDIT used as a test methodology, is a purist pursuit, and as a development methodology, is a pragmatic pursuit.

6 Conclusion

It is pointed out here that the EDIT methodology can provide a solution to the development and testing of programs in Artificial Intelligence (AI), a field where there are no sound foundations yet for either development, or testing. Also, we dare go as far as to say that EDIT may be used for the development of traditional computer science programs too.

EDIT may help in the endeavour to solve the problem of AI being an ad-hoc science. EDIT provides a methodology whereby AI can be used to develop programs in different domains and experts from those domains can be incorporated within the design and testing of such programs.

It is shown how EDIT can be used in the law domain and EDIT may address the problem brought up by Bennun and Narayanan (1991, p. 3, preface), "Could AI and law be two subjects looking for a methodology, and are

some of the contributors to this book looking for a common methodology?" One of the outcomes of EDIT is that it not only provides a methodology for testing AI programs, but it also shows how a domain itself can better get a grip on it's own methodology. This will happen at the design stage where inspectors will have many debates about the domain as well as how it can be described. Also, Bennun and Narayanan (1991, p. 4) point out: "That is 'progress' in law, computer science, and AI currently seems to take place on an ad hoc basis" and EDIT may be a methodology which, is not ad hoc, but provides a goal whereby the development of P is the goal of solving E in its initial state. The measure of success towards this goal is determined by an analysis of log files. Bennun and Narayanan (1991, Preface) point out the problems of (1) people just meeting here and there to discuss problems in constrained domains, (2) solutions not being generalisable, and (3) overcoming the lack of an agreed methodology. The EDIT methodology is one where people meet to overcome the task of analysing and formalising log files and such log files could be passed around by various researchers as direct descriptions of the problem to be solved.

Of course, we have made the assumption throughout that EDIT is a correct methodology, and that if a program passes the EDIT test then it is a valid AI program. However, EDIT does have some problems. A main problem is that it is difficult to simulate, through experiments within EDIT, software use, as it would occur in real life. Experiments can usually only approximate real life situations. An added problem is that it will be difficult to determine when a methodology like EDIT is in a state so that one can say if a program passes through the methodology, the program is a valid AI program. Therefore, at best we can only hope that methodologies like EDIT bring us closer to designing and testing AI programs that are valid. Also, it may be the case that there are other methodologies which bring AI systems closer to valid solutions, than EDIT does.

The central conclusion here is that if we are to attempt to develop computer software for standard software engineering problems, or for AI problems, then we should be doing data collection, and deciding how programs can be built in conjunction with experimentation. Any a priori decisions about how such software should be built will lead us into solutions which do not solve the problem at all. It is likely that without data collection programs for complex problems will be developed which do not solve problems adequately and hence become unreliable. Also, if AI is a science, and the

science is to progress, then a methodology which tests AI theories needs to exist. A sound methodology will reduce problems of reliability, and hence problems of legal responsibility, and will have serious implications not only for AI and law, but for AI and computer science.

7 Acknowledgements

I would like to thank Simon Morgan, Ajit Narayanan, and Derek Partridge of the Computer Science Department at the University of Exeter and Brendan Nolan from University College Dublin for providing comments on this work.

8 References

Al-Kadurie, Osama and Simon Morgan (1989) *The PCMATH System: empirical investigations*. In the Journal. of Artificial Intelligence in Education, Vol. 1(1), Fall 1989.

Bench-Capon, T.J.M., G.O. Robinson, T.W. Routen, and M.J. Sergot (1987) *Logic programming for large scale applications in the law: A formalisation of supplementary benefit legislation.* In Proceedings of the First International Conference on Artificial Intelligence and Law, 190-198. Boston, MA.

Bennun, Mervyn (1991) *Computers in court: the irreplaceable judge.* In "Law, Computer Science and Artificial Intelligence", Mervyn Bennun and Ajit Narayanan (Eds.), 45-61. Norwood, New Jersey: Ablex Publishing Corporation. (Forthcoming)

Bennun, Mervyn and Ajit Narayanan (1991) *Law, Computer Science and Artificial Intelligence.* Norwood, New Jersey: Ablex Publishing Corporation. (Forthcoming)

Clark, Andrew and Kim Economides (1991) *Computers, expert systems, and legal processes: toward a sociological understanding of computers in legal practice* In "Law, Computer Science and Artificial Intelligence", Mervyn Bennun and Ajit Narayanan (Eds.), 3-32. Norwood, New Jersey: Ablex Publishing Corporation. (Forthcoming)

Connell, John L. and Linda Brice Shaffer (1989) *Structured rapid prototyping: an evolutionary approach to software development.* Engelwood Cliffs, New Jersey:Yourdon-Press Computing Series.

Dijkstra, E.W. (1972) *The humble programmer.* Communications of the ACM, 15, 10, 859-866.

Gries, D. (1981) *The science of programming.* Springer-Verlag, NY.

Hayes-Roth, F., D.A. Waterman and D.B. Lenat (1983) *Building expert systems.* Reading, MA: Addison-Wesley.

Hempel, C. (1966) *Philosophy of natural science.* Prentice Hall.

Hoare, C.A.R. (1981) *The emperor's old clothes.* Communications of the ACM, 24, 2, 75-83.

Jackson, B.S. (1988) Editorial. International Journal for the Semiotics of Law, 1, 113-116.

Kowalski, Robert and Marek Sergot (1991) *The use of logical models in legal problem solving.* In "Law, Computer Science and Artificial Intelligence", Mervyn Bennun and Ajit Narayanan (Eds.), 99-117. Norwood, New Jersey: Ablex Publishing Corporation. (Forthcoming)

Mc Kevitt, Paul (1986) *Formalization in an English interface to a UNIX database.* Memoranda in Computer and Cognitive Science, MCCS-86-73, Computing Research Laboratory, Dept. 3CRL, Box 30001, New Mexico State University, Las Cruces, NM 88003-0001.

Mc Kevitt, Paul (1990a) *The OSCON operating system consultant.* In "Intelligent Help Systems for UNIX – Case Studies in Artificial Intelligence", Springer-Verlag Symbolic Computation Series, Peter Norvig, Wolfgang Wahlster and Robert Wilensky (Eds.), Berlin, Heidelberg: Springer-Verlag. (Forthcoming)

Mc Kevitt, Paul (1990b) *Data acquisition for natural language interfaces.* Memoranda in Computer and Cognitive Science, MCCS-90-178, Computing Research Laboratory, Dept. 3CRL, Box 30001, New Mexico State University, Las Cruces, NM 88003-0001.

Mc Kevitt, Paul and Yorick Wilks (1987) *Transfer Semantics in an Operating System Consultant: the formalization of actions involving object transfer.*

In Proceedings of the Tenth International Joint Conference on Artificial Intelligence (IJCAI-87), Vol. 1, 569-575, Milan, Italy, August.

Mc Kevitt, Paul and Zhaoxin Pan (1989) *A general effect representation for Operating System Commands.* In Proceedings of the Second Irish National Conference on Artificial Intelligence and Cognitive Science (AI/CS-89), School of Computer Applications, Dublin City University (DCU), Dublin, Ireland, European Community (EC), September. Also, in "Artificial Intelligence and Cognitive Science '89", Springer-Verlag British Computer Society Workshop Series, Smeaton, Alan and Gabriel McDermott (Eds.), 68-85, Berlin, Heidelberg: Springer-Verlag.

Mc Kevitt, Paul and William C. Ogden (1989) *Wizard-of-Oz dialogues in the computer operating systems domain.* Memoranda in Computer and Cognitive Science, MCCS-89-167, Computing Research Laboratory, Dept. 3CRL, Box 30001, New Mexico State University, Las Cruces, NM 88003-0001.

Mc Kevitt, Paul and Derek Partridge (1991) *Problem description and hypothesis testing in Artificial Intelligence.* In Proceedings of the Third Irish Conference on Artificial Intelligence and Cognitive Science (AI/CS-90), University of Ulster at Jordanstown, Northern Ireland, September. Also, in "Artificial Intelligence and Cognitive Science '90", Springer-Verlag British Computer Society Workshop Series, McTear, Michael and Norman Creaney (Eds.), Berlin, Heidelberg: Springer-Verlag. (Forthcoming)

Mehl, Lucien (1959) *Automation in the legal world: From the machine processing of legal information to the "Law Machine".* National Physical Laboratory Symposium, No. 10, Mechanisation of Thought Processes (2 Vols.). London: HMSO.

Miller, L. (1978) *Has Artificial Intelligence contributed to an understanding of the human mind? A critique of the arguments for and against.* In Cognitive Psychology, 2, 111-127.

Narayanan, Ajit (1986) *Why AI cannot be wrong.* In Artificial Intelligence for Society, 43-53, K.S. Gill (Ed.). Chichester, UK: John Wiley and Sons.

Narayanan, Ajit (1990) *On being a machine.* Volume 2, Philosophy of Artificial Intelligence. Ellis Horwood Series in Artificial Intelligence Foundations and Concepts. Sussex, England: Ellis Horwood Limited.

Partridge, Derek (1986) *Artificial Intelligence: applications in the future of software engineering*. Halsted Press, Chichester: Ellis Horwood Limited.

Partridge, Derek and Yorick Wilks (1987) *Does AI have a methodology which is different from software engineering?*. In Artificial Intelligence Review, 1, 111-120.

Popper, K. R. (1972) *Objective knowledge*. Claredon Press.

Sharkey, Noel E. and G.D.A. Brown (1986) *Why AI needs an empirical foundation*. In "AI: Principles and applications", M. Yazdani (Ed.), 267-293. London, UK: Chapman-Hall.

Susskind, Richard (1991) *Pragmatism and purism in artificial intelligence and legal reasoning*. In "Law, Computer Science and Artificial Intelligence", Mervyn Bennun and Ajit Narayanan (Eds.), 33-44. Norwood, New Jersey: Ablex Publishing Corporation. (Forthcoming)

Sutcliffe, Richard (1991) *Representing meaning using microfeatures*. In "Connectionist approaches to natural language processing", R. Reilly and N.E. Sharkey (Eds.). Hillsdale, NJ: Earlbaum. (Forthcoming)

The Byte Staff (1989) *Product Focus: Making a case for CASE*. In Byte, December 1989, Vol. 14, No. 13, 154-179.

Whitby, Blay (1991) *AI and the law: learning to speak each other's language*. In "Law, Computer Science and Artificial Intelligence", Mervyn Bennun and Ajit Narayanan (Eds.), 89-98. Norwood, New Jersey: Ablex Publishing Corporation. (Forthcoming)

Wilks, Yorick and Afzal Ballim (1991) *Liability and consent*. In "Law, Computer Science and Artificial Intelligence", Mervyn Bennun and Ajit Narayanan (Eds.), 118-131. Norwood, New Jersey: Ablex Publishing Corporation. (Forthcoming)