On the soundness and safety of expert systems

John Fox

Advanced Computation Laboratory, Imperial Cancer Research Fund, London WC2A 3PX, UK

Received August 1992
Revised November 1992

Abstract


The problems of developing sound and safe expert systems are discussed, with particular reference to medicine. The concepts, notations, methods, results and technologies which have emerged from the study of mathematical logic as a computational paradigm offer many benefits for improving the quality of expert systems. Logic programming offers a better discipline for design, specification and implementation than ad hoc development methodologies. When logic programming is combined with software engineering methods, such as a software development life-cycle, the probability of routinely developing large-scale yet efficient and sound applications will be increased. However, although soundness is a necessary property of any technology it is not sufficient for assuring safety. Established methods for improved software safety are discussed, and a number of approaches to improving the safety of medical expert systems is identified. The possibility of introducing an appropriately extended life-cycle, and the potential benefits of a formal theory of safety are discussed.

Keywords: Expert systems; logic programming; knowledge bases; Oxford System of Medicine.

1. Introduction

The introduction of AI techniques into medical informatics was motivated by a desire to increase the scope and flexibility of software and to enhance its appeal to users. By the early 'seventies AI had produced a handful of ideas with apparently radical implications for the capabilities and construction of software.

The first idea was that the distinction between algorithms and data traditionally made in computing is unnecessary and restrictive. LISP and other AI languages introduced uniform encoding of programs, functions and data as explicit structures. This led to the idea that any kind of information could and should be represented as part of an explicit, declarative, symbolic knowledge base. Symbolic methods open up capabilities which do not naturally
arise (and frequently are difficult to emulate) with conventional procedural software, such as
the ability to reason about knowledge, goals, beliefs, actions, decisions and control, and many
other kinds of meta-level reasoning (surveyed in [15]).

The development of knowledge based systems was accompanied by a growing skepticism
concerning the practicality of one of AI's traditional aims, a universal procedure for intelligent
problem solving. By the 'eighties a contrasting view, that practical problem solving programs
must be supplemented by a body of application-specific heuristics, was beginning to dominate
applications research [24]. The expertise of human experts, it was claimed, lies not in some
mysterious faculty of intelligence but in the accumulation of experience, which could be cap-
tured in a body of heuristics or 'rules of thumb'. This view culminated in the classical expert
system architecture, consisting of a 'shell' providing a few general (domain-independent) but
weak problem-solving methods, augmented by a domain-specific knowledge-base of facts
and rules.

A third observation, that certain activities or tasks seem to recur in AI applications, such
as diagnosis, monitoring, prediction and planning [41] led to the idea of domain-independent
but task-specific problem solving heuristics (e.g. Clancey’s classification heuristic, [9]) and
‘generic tasks’ [7]. These in turn led to proposals for reusable, task-oriented tools which could
be used in combination with reusable knowledge bases to provide larger building blocks for
constructing expert systems.

To summarise, the key developments in first generation expert systems were:

- **symbolic** computation, in which symbolic data-structures are used to explicitly represent
  the specialised knowledge of human experts,
- **declarative** representations of knowledge (encoding what needs to be done while leaving
to the computer the question of which algorithms to use),
- domain specific **heuristics** for problem-solving, rather than formalised algorithms,
- separation of the **knowledge base** from the **shell** that applies the knowledge in a particular
  situation,
- **shell design based on one or more generalised tasks**.

These were important innovations. They led to significant commercial products and some
success in introducing decision support systems into industrial and other settings.

The penetration of expert systems into practical medical use has, however, proved to be
much less than into commercial and industrial environments. The reasons for this relative
lack of success can be debated, but may in part be attributable to a failure to achieve clinical
credibility for the technology. Perhaps medical applications raise deep ethical issues or are
simply more difficult to build. Furthermore, the medical profession is constantly being offered
new techniques and products. Given the possible consequences of adopting unsound methods,
its standards of acceptance can be expected to be high. The special sensitivities and traditions
that surround clinical ‘judgement’ and professional responsibility have also probably played
an important part in delaying the adoption of expert systems. Given such issues one of the
weaknesses of expert systems technology may be its empirical approach to development;
the idea of using 'rules of thumb' seems intrinsically unsound and unsafe. It is perhaps
unsurprising then if doctors, and their patients, indicate a preference for professional human
judgement rather than a technology which seems to imply a surrendering of responsibility to
a poorly understood machine.
Expert systems have also been criticised from the technical viewpoints of more established disciplines. For example, decision analysts and statisticians have criticised AI methods for coping with uncertainty, and software engineers have questioned AI’s ad hoc approach to design. Though sometimes reflecting a misunderstanding of the goals of a knowledge-based approach such criticisms have raised awareness that soundness of design, clarity of specification, and integrity of implementation are issues which must be taken seriously, particularly in safety-critical fields such as medicine. The crucial components of most medical expert systems (including the knowledge base, inference engine, task management methods, and potentially the user interface) generally come without guarantees of consistency, completeness, reliability, etc.

In our attempts to address these issues at the ICRF we are pursuing an approach which we refer to as ‘logic engineering’ [16], in which we are attempting to improve the soundness of designs and implementations by combining the computational power of logic programming with the disciplines of software engineering. For example, our work on the Oxford System of Medicine, an expert system designed to support decision making in general practice, aims to develop well-defined and formally verifiable decision and inference procedures [26, 18], uncertainty management methods [30] and medical domain model [19]. Our overall goal is a methodology for developing sophisticated knowledge-based decision support systems in which technical soundness and operational safety are assured.

This paper sets out to review general issues of soundness and safety of expert systems, to draw attention to their importance and to possible ways of addressing them. In the next section, review ways by which the AIM community might win over its technical critics through a more formal approach to design and implementation is discussed. In Section 3 I turn to the question of clinical credibility, and the deeper issues of operational safety. Underlying both discussions is the suggestion that mathematical logic offers one of the most promising frameworks for achieving significant advances in knowledge systems technology.

2. Soundness and the role of logic

Logic programming (LP) can be defined as the development of software using the concepts, formalisms, methods, results and technologies which have emerged from the study of mathematical logic as a computational paradigm. The most obvious manifestation of this development is the availability of Prolog as a practical language for developing sophisticated software [42, 43], with its special virtue of having mathematical foundations that are uniquely well understood [25]. Logic programming has many strengths, and has become widely accepted in academic computer science, but it has been slower to gain general acceptance for knowledge engineering. The LP community has, until recently, tended to put less emphasis on practical engineering than on demonstrations of intellectual elegance and technical virtuosity, and techniques which are commonplace in software engineering are still relatively weak in logic programming.

However, developments in software engineering are now seen as important, and the interest in LP has received further impetus from growing emphasis on the use of formal methods for design, verification and validation of knowledge based systems, and the introduction of systematic ‘life-cycle’ methodologies for their development [35, 49]. The best known development methodology in Europe is the KADS methodology [29]. KADS is aimed at pro-
providing general tools and techniques to support design, specification, knowledge acquisition, task-oriented design etc. within a well-defined life-cycle. Life-cycle project management techniques offer improved planning, resource requirements estimation, project control and documentation, quality management, and the ability to carry out quality and cost audits [49].

Logic engineering is the use of logic programming within a formal life-cycle methodology. Taken together, KADS and the OSM span much of the logic engineering approach to developing knowledge-based systems. In the remainder of this section I shall use examples from both projects to illustrate ways in which logic-based techniques can make important contributions to design, prototyping, software specification and implementation of sound applications.

2.1. Conceptual design

The designs of first generation expert systems emphasised technologies, such as rules, frames, explanation, forward and backward chaining, and so forth. However, experience has shown that the use of purely ‘symbol-level’ techniques in expert systems needs to be guided by a ‘knowledge-level’ understanding of the applications. What this means is that, as in other kinds of engineering, designers need an understanding of the function of the components of the system as well as their form. In expert systems this implies that we develop an understanding of what the different kinds of knowledge in a knowledge base are good for and why, and not just a range of technologies to choose from. Evolution towards a higher level language of tasks, problem-solving methods, domain models and so on is becoming increasingly evident. This is not surprising; in any engineering discipline there is pressure to use standard components, since they simplify development, and may increase confidence in the integrity of the final design.

This pressure is reflected in both KADS’ and the OSM, in which knowledge bases are organised into distinct components to reflect their different functions. In KADS distinctions are made between ‘task’, ‘inference’ and ‘domain’ knowledge layers (there is also a strategic layer, but this seems to be infrequently used). There are three broad classes of task: analysis tasks (e.g. involving various kinds of diagnosis and prediction procedure), synthesis tasks (e.g. various forms of design and planning) and modification (e.g. repair and control tasks). For each task there is intended to be a generic, reusable tool; a standard component that can be instantiated with different domain knowledge (e.g. medicine) and different inference techniques. KADS provides help in designing applications by means of a standard model for structuring the knowledge base and by providing reusable task and inference methods.

The OSM is less general than KADS since it has a single generic task model. This focusses on decision making under uncertainty; it is similar to KADS’ analysis tasks, but more emphasis is placed on developing a formal theory, and design schema for decision making [17]. This aims to be more comprehensive, flexible and appropriate for knowledge-based expert systems than classical mathematical decision theory or the ad hoc methods of early expert systems [26]. As in KADS, knowledge is layered in the OSM. The first layer can be viewed as knowledge about decision making (in general) while the second layer includes more specialised knowledge about particular kinds of decision (such as diagnosis, treatment selection and test selection decisions). A third layer encodes knowledge of medicine (about diseases, symptoms, drugs, etc.) and the final layer records ‘case’ knowledge, about specific patients (Fig. 1).

There is an increasing emphasis on knowledge-based systems which exploit ‘deep’ knowl-
edge, as well as shallow empirical knowledge embodied in rules of thumb. This is commonly interpreted to mean that the knowledge embodies some sort of causal theory, but there are many other kinds of deep knowledge. Interest is growing in organising bodies of knowledge around natural 'ontologies' of the objects, processes, events, etc. in which our intuitive understanding of the world is grounded [33]. Logic offers a powerful medium for developing and formalising ontological theories, as demonstrated by the countless specialised logics in the literature. Davis provides an overview of logics which have been developed to capture the 'commonsense' meaning of a variety of ontological concepts [10]. Many of these logics, of time, space, events, action, causality, change, belief, etc. are now in a mature state and can be used with considerable confidence.

The common motivation of KADS and the OSM is that designers should be offered not just a set of implementation technologies but a higher level language of tasks, methods, general and domain-specific ontologies, etc. in which to conceive and design applications, and well understood theories to underpin their construction.

2.2. Prototyping

A feature of KBS development has been that full-scale implementation of applications is often preceded by the construction of a prototype. In some cases the principles involved in the design of the application may be unproven, or the ideas of AI or knowledge-based systems arouse skepticism among one's colleagues. Whatever the reason, there may be an understandable reticence to commit substantial resources to the development of a novel technology, and demonstration of a convincing prototype may smooth the path.

It has also been argued that the natural method of developing AI systems is the 'RUDE' (Run Understand Debug Edit) cycle, which permits requirements to be flexibly altered as empirical investigation leads to better understanding of the application. While one could not accept this as a sufficient method for developing high quality KBSs there may well be a role for this sort of iterative prototyping in the early phases of a project.

The logic language Prolog has been frequently used as a 'rapid prototyping' tool. It has a
number of features which make it appropriate for this purpose:

(a) *What, not how.* A pure Prolog program is a *declarative* description of the relationships which hold in the application domain, but since such programs also have a *procedural* reading (i.e. the program provides all the information necessary for a Prolog interpreter to derive all implied relationships) the programmer is freed from writing code to control execution. There is no guarantee that the execution will be as efficient as a program written for a specific purpose, but efficiency is not a primary concern in prototyping.

(b) *Succinctness and expressiveness.* Prolog programs are constructed directly from clauses which encode rules and facts. These naturally express some of the basic constructs of knowledge based systems and can be easily extended to express other computational metaphors without compromising the formal semantics and soundness of the underlying computational model (e.g. ideas like objects and inheritance [36]). Prolog’s declarativeness and powerful pattern matching and variable unification capabilities leads to programs that are shorter (frequently much shorter) than equivalent programs written in conventional languages.

(c) *A built-in database.* Most if not all knowledge based and expert systems require database facilities (for storing and retrieving data or knowledge, blackboard structures, etc.). A Prolog program can be viewed as a relational database extended to include deduction [20], from which facts can be retrieved directly by pattern matching or realised by executing proofs over the database. Since the programmer is freed from designing and implementing DBMS functions the construction of prototypes is substantially simplified.

(d) *Interchangeability of facts and rules.* Complex applications frequently require external utilities (such as simulation or planning utilities, or even external devices like sensors or instruments). The construction and interfacing of such utilities can require considerably more effort than the main procedure whose behaviour we want to demonstrate. In a full implementation such utilities might be encapsulated in a set of first-order rules or in external procedure calls. Whatever the implementation we can view such a procedure as equivalent to a (possibly infinite) database of its results, without altering the logic of the overall program. The prototyping effort can be reduced by including a few facts from this ‘database’. When required the full utility may be substituted for the facts without other program changes.

(e) *Meta-level expressiveness.* One of the most powerful properties of the first-order predicate calculus is its ability to express other logics. This carries over into logic programming languages in their ability to succinctly express and hence implement specialised procedures. For example, Prolog is based on a purely logical view of computation which does not naturally model time-evolving processes. However, it is straightforward to implement mechanisms with very different properties from Prolog’s own. We have made extensive use of this in a language which combines the deductive power of Prolog with the flexibility of event-triggered execution [23]. This has been used for prototyping mechanisms for simulation, planning, sensor monitoring, and so forth. Non-classical inference, execution optimisation techniques, and natural language dialogue systems are among the many other function whose implementation is simplified by the meta-level expressiveness of logic programming languages. Formal models for metalogical reasoning are now being actively investigated [2].
2.3. Logic and formal specification

The emphasis on methods based on logic is partly because of benefits in designing and implementing knowledge systems but there is another matter that I believe to be of long term importance for medical information technology. This concerns how we expect to assure the integrity of new clinical technologies.

Wyatt and Spiegelhalter have reviewed evaluation methods used in knowledge based-system development. A comparison of these methods with standard methods for evaluating medical techniques, such as clinical trials of new drugs [50] showed that testing methods need to be substantially improved. Empirical testing of expert systems certainly needs improvement, but this cannot be our only methodological goal. I would draw a different analogy, with mature engineering disciplines. Many branches of engineering have moved beyond purely empirical testing (such as test-flying aircraft prototypes) because they have established strong design theories (e.g. for stress analysis for predicting properties of the airframe, laminar flow models for predicting turbulence and aerodynamic behaviour). The consequence is that designers can confidently predict failure modes, performance boundary conditions and so forth before the systems are implemented. The formulation of a comprehensive design theory, which guides the design process and provides a basis for predicting the behaviour of specific application systems, seems to be an important objective for expert-systems research.

A promising approach to formalisation may be to use well-defined specification languages and verification procedures. Van Harmelen and Balder [46] summarise advantages of using formal languages for describing the structure and/or behaviour of software systems as:

- the removal of ambiguity,
- facilitation of communication and discussion,
- the ability to derive properties of the design in the absence of an implementation.

Logic offers an expressive language for capturing different kinds of knowledge in different functional layers. Not only is the predicate calculus (and the logic programming language Prolog) sufficiently expressive to capture the different types of knowledge, it can also serve the role of a specification language for formalising layers as (reusable) domain theories, task theories, and so on. Two different specification tasks need to be undertaken for the classical expert system architecture: one for the 'shell' and one for the knowledge base. These specifications may be rather different and may require different tools, though logic can contribute substantially at both levels.

At the level of the inference engine, for example, the use of careful specification techniques can reveal a lack of soundness in a design. Some time ago I programmed a truth maintenance mechanism for use in a prototype expert system. A colleague, Paul Krause, examined the properties of this mechanism and showed that under certain circumstances the TMS was unsound and could lead to pathological reasoning. A modified truth maintenance system was specified using a formal specification language [31] and the consistency and effectiveness of the specification was formally proved [32].

The use of formal techniques for specifying knowledge bases is not so straightforward, since domain knowledge may be poorly defined and incomplete. However, recent proposals for formal specification languages which are specifically adapted for knowledge base design seem promising [48]. One such language has been developed within KADS. This language (ML)^2 provides constructs for viewing knowledge bases as organized collections of logical
'theories', for communication between theories and for describing meta-level operations on theories [46].

Theories in a logic consist of a set of formal terms (sentences) which are to be interpreted according to the inference rules of the logic. A theory in first-order logic (FOL) consists of a set of facts and implication rules to be interpreted according to the rules of the propositional calculus extended with quantifiers. Theories in (ML)² are essentially equivalent to theories in FOL together with meta-information about the kinds of objects that the theory mentions. Loosely speaking, each task, inference technique or body of domain knowledge can be specified as a theory in (ML)². The formalism provides a language in which such theories can be specified in a modular reusable way.

A theory in (ML)² consists of a signature which defines a language plus a set of axioms expressed in this language. The general structure is presented below (this is not a formal presentation so I have slightly simplified the syntax):

```
theory (name)
  import (set of theory names)
  signature classes (class hierarchy)
    constants (set of constants and their classes)
    functions (set of class → class mappings)
    predicates (set of predicates and arguments)
  variables (set of symbols and their classes)
  axioms (set of facts and rules)
end-theory
```

A knowledge base consists of a collection of theories and sub-theories. Theory definitions can assume (import) the definitions of other theories. This clarifies the structure of the knowledge base and avoids repetition if a sub-theory is reused. The signature of a theory defines the syntax to be used in expressing axioms of the theory and a collection of integrity constraints on them; the facts and rules of a traditional knowledge base (or Prolog program) would form the axioms of a theory in (ML)². If axioms include first-order rules (i.e. rules including variables) the classes of values the variables can have is also specified. A concrete example of (ML)² being used to define a simple layered structure similar to that of the OSM is given in Fig. 2. The three theories express a fragment of knowledge dealing with the diagnosis of disease from qualitative knowledge of the causal relationships between symptoms and signs and diseases. The diagnosis procedure is a simplification and specialisation of a more general symbolic decision procedure [26]. The definition assumes (imports) a medical layer (here just a set of causal facts) and a patient record (a set of findings).

Specification techniques for knowledge engineering need not be restricted to classical logic. Krause et al. made use of modal logic in specifying their revised truth maintenance system; Van Harmelen and Balder propose the use of quantified dynamic logic for specifying complex procedures and tasks (an extension of FOL which permits expression of programming concepts like sequences of operations on states, iteration, etc.), and the diagnosis procedure in Fig. 2 assumes that 'arguments' are constructed using LA, a logic of 'argumentation' described in [18].

Van Harmelen and Balder remark that "if 'knowledge engineering' wants to live up to
its name and is to become a proper engineering activity, a similar development of the field towards a formal treatment of the models it is concerned with must take place”. Formal specification languages are intended to enable software engineers to produce a mathematical model of the software they wish to build before committing themselves to coding it. As well as providing an unambiguous statement of the program’s intended behaviour, the specification may be subject to mathematical proofs and analysis of its properties, helping to avoid potentially disastrous errors. This is not to say that formal specification is a panacea, since use of formal specification is difficult on large scale applications and there are certain technical restrictions on its use, but it offers a promising discipline. Tools for simplifying the use of specification methods are under active development.

2.4. Implementation

Implementation of an efficient application raises issues that are not addressed by prototyping.
and specification techniques. Indeed, a successful prototype has often fostered a false sense of security about the ability to deliver a practical application. For some applications, such as simple diagnosis systems, full implementation may require only refinements of the original prototype. Frequently, however, the prototype may only incorporate part of the functionality and/or knowledge of the intended system; its function is solely to establish, or demonstrate to others in an organisation, that the application is viable in principle. Unfortunately, however, it has frequently been found that developing a prototype expert system into a practical delivery system proves more problematic than expected; the pilot application does not 'scale up'.

Scaling up may raise problems for ensuring the integrity of the knowledge base, or achieving acceptable efficiency when the application is reasoning over a large knowledge base. These are both significant challenges to the Oxford System of Medicine. The OSM is concerned with general practice and primary care, and general practitioners have to deal with a wide range of tasks and a very large body of general medical knowledge. (We estimate that to cover a substantial proportion of general medicine the OSM knowledge base could require a domain theory equivalent to around 10M facts.)

(a) **Integrity.** An important goal of the KADS and OSM projects is the development of knowledge acquisition tools which include verification facilities. The (ML)\(^2\) formalisation provides a basis for some automated verification; TheME editor makes use of the syntax information in the signature of an (ML)\(^2\) theory to check that knowledge updates are consistent with the signature and to guide extensions to the knowledge base [4]. Checking the content of axiom sets requires different techniques. There is now a considerable literature on checking of knowledge bases for coherence and consistency [8] and for identifying anomalies in the domain knowledge, where logic provides a clear semantics for anomalies [40]. Second-generation knowledge based systems, in which emphasis is placed on the use of models in the knowledge base (such as causal models for diagnostic systems), can also benefit from logical verification procedures [22] as can reasoning about the integrity of temporal information [6]. Taking a different but similarly motivated approach the OSMOTIC editor uses a generalised model of medical concepts and their interrelationships to detect model violations and suggest repairs to the OSM domain layer [19].

(b) **Efficiency.** AI tools have acquired a (rather exaggerated) reputation for inefficiency, but use of very general representations and meta-level programming techniques does tend to entail heavy computational costs, and this may be unacceptable for large applications. However, recent work has led to a number of techniques for improving efficiency without leaving the security of the logic programming model. Inefficiency of logic programs is commonly due to recomputation resulting from unnecessary backtracking during search. Partial evaluation is an optimisation technique whereby unnecessary computation is avoided by compiling a very general Prolog program into a more specific and more efficient one [25]. Meta-level or 'piggy back' interpreters can be used to exploit specific control knowledge by dynamically reordering goals during execution of Prolog programs [38], and in constraint logic programming efficient constraint solving algorithms are substituted for classical backtracking search [47]. Commercial systems are becoming available that combine efficient database management techniques with the power of FOL, facilities for integrity constraint checking, fast run-time constraint solving, and the ability to run on parallel machine architectures. This combination of capabilities is likely to
have considerable implications for our ability to deliver complex, yet efficient and sound applications.

2.5. Fielding

Introducing novel software into established work settings commonly demands tolerance and flexibility from users and imposes changes to work practices, and organisational disruption. Adopting a formal methodology for software design and development does not solve these problems but there are indirect benefits from carefully articulated design, the opportunity of staff to try out (and participate in the development of) prototypes, and enhanced quality and reliability of delivered software.

User participation in design is always good practice of course, but the principal challenge to the software designer, particularly in a medical context, is to achieve the trust of those affected by the new techniques. Demonstration of effectiveness, using proven clinical evaluation techniques will foster trust, and good design facilitates convincing evaluation, but even these are insufficient. The most stringent empirical testing cannot prevent the occurrence of unanticipated events or side-effects of use – and consequent disasters. Proving that software satisfies formal requirements and complies with a software specification cannot help much, since the requirements or specification may be faulty.

Given the controversies that still surround the use of AI a crucial question is can we make expert systems provably safe in some sense? Traditional safety engineering techniques for electromechanical systems have proved to be capable of reducing risk of failure to extremely low levels. Unfortunately software raises new and severe problems, and decision support systems exacerbate these problems. The fundamental difficulty is that safety engineering has been primarily concerned with component failure, whereas decision support software may operate perfectly yet has unacceptable downstream consequences. In the next section I review some of these difficulties in more detail, and explore some possible directions for development.

3. The safety of expert systems

Much medical software is ‘safety-critical’, meaning that errors in operation or use can lead to death or injury (and legal liability). To pick just one example, problems with the over/underdosing of patients receiving radiotherapy have been caused by software design errors [44].

Decision support systems can behave unsafely in a number of ways. The software may fail to:

- detect a hazardous situation (e.g. a possible disease),
- recommend an appropriate action (e.g. an effective treatment),
- predict a hazardous side-effect of some recommended action (e.g. a potential drug interaction),
- recommend an action to prevent a latent hazard turning into a disaster (e.g. post a veto on using drugs which could interact with a prescribed drug).
The increasing use of AI techniques in decision support and other systems, particularly in the context of autonomous systems, has attracted significant concern. The problems with the radiotherapy machine arose in the context of conventional software, for which there is not only vast experience but also many techniques for assuring software quality. Since our understanding of the behaviour and consequences of using expert systems is limited we cannot be complacent about their use in safety-critical applications. As one author succinctly puts it, one of the problems with software design and development is that we have been overly concerned with what a system shall do, and insufficiently concerned with what it shall not do. It is hardly surprising, therefore, that there have been calls for restrictions on the deployment of unsupervised or autonomous systems in safety-critical applications [5] and for keeping a human operator 'in the loop' in medical applications [37].

Even if we restrict our attention to advisory expert systems (under the supervision of professional clinical staff) there are still reasons for concern. We have discussed how formal design and development techniques can help to ensure soundness and, hence, quality of software. Soundness is a necessary condition for safety, but unfortunately it is not a sufficient condition. Soundness is a technical property which ensures that a program complies with a formal specification, but it does not ensure that the original specification is correct or that the consequences or side-effects of using the software are only those that are intended by the designers and users.

Classical safety engineering developed within aerospace, process control, power, and other industries involving complex electromechanical systems. It has been extraordinarily successful. If established safety engineering methods are adopted designers can be confident of achieving extremely high reliability levels. Requirements for system failure rates of the order of $10^{-9}$ over years of operation are not untypical. Perhaps we should look at engineering practices in such domains for techniques we can use in expert systems?

3.1. Traditional strategies for minimising disasters

Safety engineering has been traditionally concerned with 'faults' and 'hazards'. A fault is a failure to behave according to the intentions of the designers (for which the specification is a partial indication) and a hazard is a situation which might lead to a total or partial failure having death or injury (etc.) as a consequence. Faults do not necessarily lead to hazards, and hazards to not necessarily lead to disasters (e.g. near-misses of aircraft) but potential errors need to be understood and prevented. Traditional safety engineering therefore involves two main kinds of activity:

1. analysing faults and the hazards they give rise to, and
2. incorporating techniques for minimising the likelihood of faults and for preventing hazards turning into disasters.

The need to design quality into software from the beginning of the development life-cycle is now generally accepted. An analogous view can be taken of software safety: "The goal of system safety is to design an acceptable safety level into the system before actual production or operation. ..." [34]. The adoption of life-cycle methodologies has helped to improve the quality of conventional software and promises to do the same for knowledge-based systems. Figure 3 suggests an approach in which a safety life-cycle is executed in parallel with the quality life-cycle. The remainder of this section considers the activities composing this life-
cycle, considering the relevance of established safety techniques, and ways in which they might be enhanced to address the special challenges of expert systems.

Fig. 3. A safety life-cycle should be adopted in parallel with the normal development life-cycle.

3.1.1 Hazard analysis

Before formulating a safety strategy for a proposed system it is necessary to establish a clear picture of the circumstances that may lead to hazards. Among the techniques traditionally used are fault likelihood estimation and hazard analysis.

When designing any new equipment it is normal to use standard components wherever possible. For safety-critical systems this may have the great advantage that historical failure data are available for such components. Failure data provide a basis for estimating the likelihood of operational faults (and collections of faults) and hence to predict the expected reliability of the total system.

As we have discussed there is now considerable effort to develop reusable software components in software engineering and increasingly in knowledge engineering as well. However, in the context of safety the analogy with standard physical components does not take us very far. It does not make much sense to collect historical data to assess the ‘reliability’ of software components. Ideas like ‘mean time between failure’ which are so informative about electromechanical systems are not applicable to software. Software does not fail for the same
sorts of reasons as physical devices do (e.g. wear and tear). If a fault is manifested we call
this a bug and fix it, so it should not happen again in that form. Indeed it is not currently
feasible to arrive at a meaningful estimate of the likelihood of a software component ‘failing’
at all [34, 39, 1].

Another procedure for analysing likely sources of hazard involves the systematic explo-
ration of those faults that could conceivably arise during system operation, possible inter-
actions between faults, and interactions between faults and environmental events. These
methods are part of ‘hazard analysis and risk assessment’, and frequently involve building a
model of all the causal dependencies in the proposed system, its failure modes, etc.

Hazard analysis is naturally helped by a systematic approach but in the end its effectiveness
depends on the imagination and assiduousness of the design team. With software the problems
seem to be much worse because of the abstract nature of software processes, the ease with
which systems can be made very complicated and the awesome potential of software to
manifest unexpected interactions. Hazard analysis can be considerably helped by constructing
prototypes and/or simulating the use of the software to reveal faults, software errors and so
on. However, it is difficult to provide realistic test conditions and there is no guarantee that
the prototype or simulation is accurate or complete.

In short, there are no techniques which can guarantee that all possible faults and hazards
are exhaustively identified for any but the most trivial software, nor techniques for reliably
estimating the relative likelihood of hazardous situations. Strategies for preventing and min-
imising the consequences of hazards must, in some way, take this into account.

3.1.2 Hazard avoidance

The engineering ideal is to be able to design a system so well that problems simply cannot
occur. Formal specification, refinement and verification of software can substantially improve
the integrity of a program, as can the use of formally verified standard components, such as
knowledge bases and specialised theorem provers. However, there are severe limitations on
the capability of formal design techniques to completely prevent hazardous situations from
arising. Current formal design methods are difficult to use and time-consuming, and may
only be practical for relatively modest applications. Even if we reserve formal techniques
for the safety-critical elements of the system we have seen that the soundness guaranteed by
these techniques can only be as good as the specifier’s ability to anticipate the conditions and
possible hazards that can hold at the time of use.

These problems are difficult enough for ‘closed systems’ in which the designer can be con-
fident, in principle, of knowing all the parameters which can affect system performance and
design the software to respond to abnormal states or trends. Unfortunately all systems are to a
greater or lesser extent ‘open’; they operate in an environment which cannot be exhaustively
monitored and in which unpredictable events will occur. Furthermore, reliance on specifi-
cation and verification methods assumes that the operational environment will not compromise
the correct execution of software. In fact of course software errors can be caused by transient
faults causing data loss or corruption; user errors; interfacing problems with external systems
(such as databases and instruments); incompatibilities between software versions; and so on.
The clinical environment is a prime example of an open environment in which the carefully
formulated recommendations of an expert system may be quickly compromised by unforeseen
interactions and side-effects.
3.1.3 Fault removal

Hazards, and consequent disasters, cannot be totally avoided even with the best intentions and best design practices. Inadequate design foresight, including specification errors, has been found to be the greatest source of software safety problems [13, 21]. The use of formal specification techniques has not removed the need for rigorous testing and debugging of conventional software [39]; the adoption of analogous techniques in AI systems is similarly unlikely to remove the need for empirical testing.

The need for rigorous clinical trials of new medical technologies is generally accepted. However Wyatt and Spiegelhalter have raised serious doubts about testing in practice: “only about 10% of the many medical knowledge-based systems that have been described over the years have been tested in laboratory conditions, while even fewer have been exposed to clinical trials. This may be because systems were built to investigate certain tools or techniques, or because of lack of resources, or it may be because of confusion about what to test, and how to test it” [50].

Even if the best available methods for empirical evaluation of the performance of an expert system are employed prior to installation, this will not remove all residual sources of difficulty. Even if we follow the best practice for clinical trials we will not fully address the safety issue. ‘Live testing of computer responses to catastrophic situtions is ... difficult in the absence of catastrophes.’ Conventional systems have provided reliable service over long periods only to fail disastrously (for the patient) in unanticipated conditions of operation or use (e.g. the radiotherapy example) and there seems to be no reason to think that current decision support systems will be any different. More techniques are clearly needed; one possible strategy is to build some sort of fault tolerance into the design.

3.1.4 Fault tolerance

If it is not possible to avoid all errors perhaps it is possible to manage them when they occur. A proven technique with mechanical and electronic systems is to build tolerance into systems. The classic fault tolerance technique involves redundancy, in various forms. Mechanical systems may be duplicated so that if one fails others are likely to remain operational, for example.

Analogous techniques have been explored for software design. Simple duplication of software is of course unlikely to be effective, since identical programs will have identical error behaviour. However, so-called ‘N-version’ techniques, in which a number of different versions of the software are run simultaneously have been discussed (e.g. [1]). Assuming the different version have different failure modes, supervisory software can resolve conflicts by letting the different programs ‘vote’ and act on the majority conclusion. This idea might be applied to knowledge-based systems. For example, in diagnosis one might compare the conclusions from applying statistical reasoning with those from deterministic causal simulation and with reasoning from ‘textbook’ cases, with a meta-level component resolving any conflicts that arise. Unfortunately we cannot be complacent that introducing redundancy into knowledge-based systems will necessarily substantially reduce the likelihood of problems. Knight and Leveson were unable to find independence of failure behaviour between independently produced versions of software [28], and there is as yet little evidence that different reasoning techniques have distinct error characteristics. This is illustrated by a study of medical expert systems which compared the performance of two very different methods in diagnosing abdominal pain. 50 patients who were known to fall into one of 5 clinical
categories (non-organic or functional disease, cholecystitis, duodenal ulcer, gastric ulcer or cancer) were classified from clinical symptoms and signs into these categories using

(a) a Bayesian calculation of probability, and
(b) a set of logical rules.

As Fig. 4 shows the error patterns produced by the two apparently different techniques were almost identical [14].

<table>
<thead>
<tr>
<th></th>
<th>F</th>
<th>Ch</th>
<th>DU</th>
<th>GU</th>
<th>Ca</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>5</td>
<td>4</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ch</td>
<td>1</td>
<td>7</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>DU</td>
<td></td>
<td></td>
<td>8</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>GU</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>Ca</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

(a)

<table>
<thead>
<tr>
<th></th>
<th>F</th>
<th>Ch</th>
<th>DU</th>
<th>GU</th>
<th>Ca</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>4</td>
<td></td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Ch</td>
<td>1</td>
<td>6</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>DU</td>
<td>1</td>
<td>9</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GU</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Ca</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>8</td>
</tr>
</tbody>
</table>

(b)

Fig. 4. Similarity of diagnosis patterns of rule-based and statistical diagnosis methods [14]. The objective diagnosis is on the vertical axis and the computer diagnosis on the horizontal. Dots in (b) indicate that there was a tied assignment.

3.2. Towards a formal safety theory

From the foregoing one might conclude that using a basket of safety methods (hazard analysis, formal specification and verification, rigorous empirical testing, fault tolerant design) will significantly decrease the likelihood of hazards and disasters. However, there is at least one weakness common to all these methods. They rely on the design team being able to make long-range predictions about all the clinical circumstances that may hold when the system is in use. This is unrealistic, if only because of the countless interactions that can occur. The
problem is well illustrated by the OSM. It is conceivable (just) that all hazards that could arise in specialist clinical management could be anticipated, but in general practice it is common that patients suffer from multiple, independent problems, with concurrent treatments in progress. Indeed different specialist clinics may be taking primary responsibility for managing certain conditions. The scope for unforeseeable interactions is vast.

To address this problem provision might be made for developing a dynamic safety strategy to complement the static techniques described. Here the expert system would be assigned a task of reasoning about safety issues in parallel with its principle tasks of diagnosis, test and treatment selection, etc. An interesting illustration of this idea in a non-medical domain is suggestive. The ‘safety bag expert system’ [27] was developed for managing routing of rolling stock through a rail network. Simply planning the shortest or some other least-cost route through the network is clearly hazardous. A section of the route may have other wagons on it, or certain points (switches) might be set such that another train could enter the section, for example. The safety bag expert system has a dual channel design in which one program proposes viable routes through tracks and points while a second system monitors the proposed routes for potential hazards. The latter system is a rule-based system in which the rules’ conditions embody knowledge of safety regulations and their actions are to veto (or commit) proposed routes. Examples of rules for vetoing and committing proposed rail routes used in the safety bag expert system are:

- **IF** there is a route element of shunting route  
  **AND** the new route is a train route  
  **AND** the start signal of the shunting route is not contained in the new route  
  **THEN** the route is not admissible

- **IF** there is a request to perform the global locking check  
  **AND** all switches in the route are correct position  
  **AND** all switches in the route are interlocked  
  **AND** all switches in the route are flank protected  
  **AND** the status of the route is admissible  
  **THEN** commit the request and change the status to locked.

The two main hazard points for expert systems are

1. **failure to maintain continuous data surveillance and anticipation of the effects of proposed actions, and**
2. **failure to take or recommend appropriate action in the light of these assessments.**

For **surveillance** an established technique is to introduce ‘watchdogs’; such as rules for detecting the simultaneous prescription of incompatible medications. Current watchdog techniques use purely shallow reasoning; research into deeper surveillance techniques is required. For example we need general techniques for detecting possible hazards resulting from a clinical action taken for one purpose compromising some other completely independent clinical objective. Dynamic hazard analysis might also include a process of forward projection of
clinical recommendations to establish whether there are unacceptable 'possible worlds' which could result from a recommendation being followed. Candidate techniques for possibilistic projection are qualitative simulation [12], model-based prediction [45] and reasoning methods employing possibilistic logics [11].

We also need to explore new ways of deciding how to act in hazardous situations. The safety bag expert system approached this in terms of rules for vetoing routes through the rail network, where the rules embody safety regulations. A simple veto is something of a blunderbuss; it may be desirable to have reasonable assurance that an expert system will fail safe (if use of a drug is contraindicated then the user must be told) but we may also be willing to accept that it 'fails soft'; if reasoning methods yield contradictory diagnoses then a management plan that covers for both possibilities is required (the generic antibiotic example was an instance of this).

The rules in the safety bag represent application-specific safety conditions. Generalising this idea one might consider extending the inference methods and theories of a clinical decision support system to include an abstract safety theory, which sets out in general terms what is permitted and what is not. An illustrative domain-independent 'safety theory' specified in (ML)$^2$-style is given below.

theory safety
  import application-theory
  signature
    classes states(safe unsafe) actions authorities
  functions
    simulate: actions × states → states
  predicates
    unsafe: unsafe
    permitted: actions
    obligatory: actions
    proposed: actions × states
    possible: actions × states
    authorised: authorities × actions
  variables Act: actions State,New: states Auth: authorities
  axioms
    simulate(Act,State) = New
    → possible (Act,New)
    proposed(Act) ∧ not(possible(Act,New) ∧ unsafe(New))
    → permitted(Act)
    proposed (Act) ∧ possible(Act,New) ∧ unsafe(New)
    → obligatory(authorise(Act))
    obligatory(authorise(Act)) ∧ authorised(Auth,Act)
    → permitted(Act)
end-theory
The basic idea behind the theory is that if an action is proposed and the system cannot project a possible hazard arising from the action then it is permitted to take that action, but if there is a possible hazardous consequence then approval by an empowered authority is obligatory. The specification assumes that possibilistic projection is carried out by a simulation function. This function, as well as the details of safe/unsafe states, actions and authorities would be specified in an imported application-theory. The signature places constraints on the definitions of the classes, functions and predicates in the application-theory that can instantiate the general safety regulations. Mathematical logic and AI offer some appropriate tools for formalising the regulations in such theories, such as deontic logic, the modal logic of permission and obligation.

Whatever form it eventually takes we would like a ‘safety theory’ to be a reusable component, providing a general structure which can be used to verify the construction of application knowledge bases, and guide their use in dynamic hazard management. Naturally this is speculative, but it is really only an elaboration of established approaches to improving safety in conventional software, in which software layers specialised for event and trend monitoring and for effecting panic operations, are integrated into the design.

4. Conclusions

Almost two decades’ experience of building expert systems has shown that strong methodologies are needed to ensure high quality systems. I have tried to make a case that the concepts, notations, results, techniques and technologies of logic programming combined with software engineering methods offer a potent discipline. Not only have logicians arrived at a deep understanding of logical formalisms during the long history of their development, but contemporary work in AI has yielded powerful and practical tools that embody their results. From such observations I have tried to demonstrate the potential of a systematic logic engineering approach to the design and implementation of verifiably sound expert systems.

However, if we are to achieve trust in any medical technology it is not enough to prove soundness. We must also demonstrate, so far as is possible, that the technology is safe. Alongside the pursuit of rigorous development techniques, therefore, we also need to invest effort in investigating general safety principles, together with formalisms and practical design techniques which incorporate them. Some techniques may prove to be straightforward extensions of existing safety engineering methods, such as a safety life-cycle, but AI techniques also suggest novel ways of interpreting traditional ideas and the possibility of developing a dynamic safety strategy as well.

One sometimes hears that ‘we are trying to improve current clinical results, not achieve perfection’. There is some comfort in such observations, particularly for a technology that still has to prove itself, but medical informaticians should surely not be satisfied with incremental improvements. If there is any chance of emulating the ultra-high safety record achieved in other industries then that should surely be the goal we set for ourselves.

Acknowledgements

I am grateful to Peter Hammond, Frank van Harmelen, Jun Huang, Isaac Khabaza, Paul
Krause, Serdar Uckun, and an anonymous referee for many comments that have helped to improve this paper.

References

On the soundness and safety of expert systems