Socrates: a flexible toolkit for building logic-based expert systems

R Corlett*, N Davies*, R Khan*, H Reichgelt† and F van Harmelen†

This paper describes the architecture of the Socrates toolkit for building expert systems. The authors analyse the problems associated with existing expert system tools and propose a solution based on the use of logic and meta-level inference. The abstract architecture for the toolkit is described which embodies this combination of logic and meta-level inference. This architecture can be instantiated to create a system that is specialized for a particular application. This specialization process can be seen as a methodology for building expert systems. The three stages of this methodology are discussed in detail, along with descriptions of how the Socrates toolkit supports it. The current implementation of Socrates, plus a number of applications of the toolkit are described, and the open problems are discussed.

Keywords: expert systems, toolkit, architecture, building methodology

Most of the tools that are currently available for constructing expert systems fall into two categories, namely expert system shells and high level programming language environments. Both of these types of tool suffer from a number of shortcomings which limit their usefulness.

The first type of available tools, 'shells', are usually constructed by abstraction from an existing expert system. Thus, a shell normally consists of an inference engine and an empty knowledge base, and some debugging and explanation facilities. Buyers of a shell often believe, and manufacturers often claim, that the shell is appropriate for a number of different applications. However, a large number of people have expressed dissatisfaction with expert system shells. The two most frequently heard complaints (e.g. Alvey[1]) are first, that the view that the inference engine which was successful in one application will also be successful in other applications is unwarranted, and second, that the knowledge representation scheme often makes the expression of knowledge in another domain awkward, if not impossible. For example, a production rule language that was designed for solving a classification problem using backward chaining would probably not be very suitable for solving a design or planning problem.

On the other hand, proponents of high level programming language environments (sometimes known as 'hybrid systems'), such as LOOPS[2], KEE[3] or ART[4], can be seen as taking a more pluralistic position: instead of providing knowledge engineers with a single prefabricated inference engine, one provides them with a large set of tools, each of which has proven useful in other applications. LOOPS, for example, provides object-oriented programming with inheritance, a production rule interpreter and active values, as well as Lisp.

While we accept that hybrid systems are useful as tools for program development, we would claim that they are less useful as tools for building expert systems. Their main problem is that they provide the knowledge engineer with a bewildering array of possibilities, and little, if any, guidance as to the circumstances in which any of these possibilities should be used. Unless used by experienced programmers, high level programming environments encourage an *ad hoc* programming style in which no attention is paid to a principled analysis of the problem at hand to see which strategy is best suited for its solution.

The conclusion that is drawn from the problems associated with shells and high level programming language environments is that a number of different 'models of rationality' are needed, and that different applications require different models of rationality. When constructing an expert system, the knowledge engineer then has to decide which model of rationality is appropriate to the application at hand. A model of rationality has both a 'static' and a 'dynamic' aspect. These two aspects correspond to the knowledge about

*GEC Research, Marconi Research Centre, Chelmsford, UK
†Department of Artificial Intelligence, University of Edinburgh, Edinburgh EH8 9YL, UK

0950-7051/88/030132-11 $03.00 © 1988 Butterworth & Co (Publishers) Ltd

the application area plus a strategy that describes how to use this knowledge when solving a problem. The 'interpretation models' from Breuker and Wielinga[5] correspond closely to our models of rationality.

The distinction between the static and the dynamic aspects of a model of rationality corresponds to a distinction one can make between two different aspects of an expert system. First, there is the 'domain' in which the expert system is to solve problems. For example, the domain of an expert system may be electronics, or internal medicine. Secondly, there is the 'task' which the knowledge engineer wants the expert system to perform. For example, the task of a system can be diagnosing a faulty electronic circuit or designing a new circuit. It is interesting to note that the problems with shells reflect these two aspects of an expert system. The first complaint about shells concerning the expressiveness of the knowledge representation language is related to the structure of the domain. The second complaint concerning the rigidity of the inference engine is related to the task of the expert system. As pointed out by Chandrasekaran[6], typical expert system tasks such as diagnosis, planning, monitoring, etc. seem to require particular control regimes.

Socrates allows the use of a variety of logical languages and a variety of control regimes to solve the problem of the lack of flexibility associated with shells. By using logic as its unifying framework, and by providing guidelines for the choice of both the representation language and the control regime, Socrates avoids the unstructured richness of the hybrid systems. These points are discussed in detail below.

USING LOGIC FOR KNOWLEDGE REPRESENTATION

Socrates uses a 'logical language' as the main formalism to implement the static aspects of the required models of rationality for different domains. On the one hand, logical formalisms are rich enough to provide different models of rationality, while on the other hand the use of logic provides a unifying framework for the system which saves it from the unstructured richness of the hybrid systems. This choice of logic as the main formalism implies that logical languages will serve as the representational scheme, while logical deduction will be the paradigm for the inference engine.

Many advantages come with the use of logic as the main knowledge representation formalism. Logic comes with a formal semantics, it has well understood properties regarding completeness, soundness, decidability, etc., and it has great expressive power. For a further elaboration on these arguments, see Corlett, Davies, Khan, Reichgelt and van Harmelen[7].

USING META-LEVEL INFERENCE FOR CONTROL

The correspondence between the two aspects of a model of rationality and the problems with shells suggests that a model of rationality should be computationally realized as a knowledge representation formalism plus a control regime for using this formalism. A number of arguments can be given for the explicit and separate representation of control knowledge. First, a system with

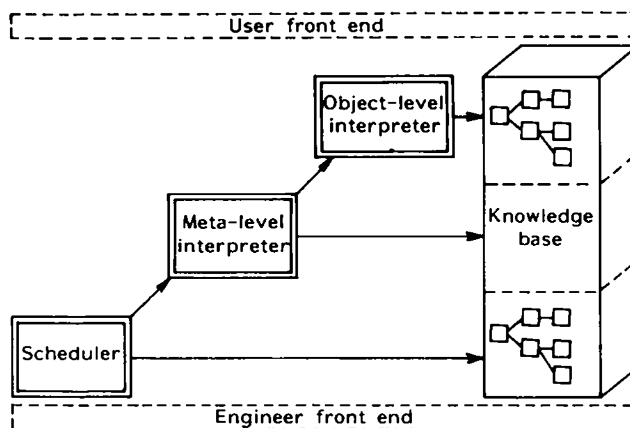

Figure 1. Socrates architecture

an explicit representation of its control strategies is easier to develop, debug and modify, as argued by Davis[8], Bundy and Welham[9], Clancey[10] and Aiello and Levi[11]. Second, the separation of control knowledge from domain knowledge (or 'object-level knowledge') increases the reusability of the system: the same object-level knowledge can be used for different purposes, and the same control knowledge can be used in different domains[5,10,12]. Finally Clancey[13] stresses the importance of explicit control knowledge for the purpose of explanation.

The separation of control knowledge from domain knowledge allows domain knowledge to be purely declarative in nature. While formulating domain knowledge we do not have to worry about efficiency, only about the 'representational adequacy' (in the words of McCarthy). In the control knowledge, on the other hand, the efficiency of the problem solving process is the most prominent aspect ('computational adequacy').

ABSTRACT ARCHITECTURE OF SOCRATES

As argued in many other places in the literature (see [10,12,14-20] among others), an architecture with a separate object-level and meta-level interpreter can be used to implement the required separation of domain and control knowledge. This architecture is shown in Figure 1. The two-layer architecture of object-level and meta-level interpreter is extended in Socrates with a third level, the scheduler, discussed below. Each of the three interpreters communicates with the knowledge base in order to store and retrieve logical propositions. As described below, the knowledge base can be organized into a number of partitions, each of which can be hierarchically organized into subpartitions.

The Socrates architecture distinguishes between front-ends to be used by the 'end-user' and the 'knowledge engineer'*. Knowledge engineers and end-users will need different tools for communication with the system. For example, an end-user, when asking for an explanation, will not want to see the entire proof tree, but rather 'edited highlights'. A knowledge engineer, on the other hand, may want to be able to monitor the reasoning

*The end-user is the person who uses the expert system application built with the Socrates toolkit, whereas the knowledge engineer is the person building such an application system.

process much more closely and may require the full proof tree. Similarly, a knowledge engineer will need tools for adjusting the behaviour of the interpreters and for editing the knowledge base, whereas an end-user only needs to browse in a read-only manner through the knowledge base. Furthermore, the levels of abstraction at which the system communicates will differ between end-user and knowledge engineer.

This general, application independent architecture needs to be 'configured' for a particular expert systems application, given the characteristics of the domain and the task of that application. The use of logic as the underlying framework for the system gives a very specific meaning to the process of configuring the system for a particular application. Rather than having to choose between arbitrary representational schemes and inference methods (as is the case in the hybrid systems), the configuration process now consists of three well defined stages; i.e. the declaration of a 'logical language', the declaration of a 'proof theory' for that logical language, and the declaration of a 'proof strategy'.

LOGICAL REPRESENTATION LANGUAGE DECLARATION

As the first step in the process of configuring a system for a particular application, the knowledge engineer defines the logical language that will be used for representing the domain knowledge. In the context of a knowledge-based system, where the logical language is to be used for representational purposes, the 'syntactic' declaration of the language has to be augmented with a mechanism for 'storage and retrieval'. Each of these aspects will be discussed in this section.

Representation language

As the first part of the declaration of the logical representation language, the knowledge engineer has to declare the 'vocabulary' of the language. The predicate symbols, function symbols and constants that are going to be part of the representation language have to be defined.

Socrates uses 'many-sorted logics' of the kind proposed by Walther[21], making use of typed quantification, where the sorts are organized in a hierarchy. Each sort is used to represent a non-empty set of individuals belonging to that sort. In such logics the sort hierarchy is defined by a partial ordering on the set of sorts, corresponding to a lattice structure with the universal sort as the maximal, and the empty sort* as the minimal element in the set. This lattice is more general than a tree structure, since it allows sorts to have more than one supersort, thereby increasing the expressiveness of the sort of system. The use of many-sorted logics over unsorted logics brings a number of advantages. First, complicated unsorted expressions can be reformulated in a much simpler many-sorted form. For example, the unsorted expression:

$$\forall x \exists y [\text{human}(x) \rightarrow \text{human}(y) \;\&\; \text{mother}(y, x)]$$

can be rewritten, after the declaration of human as a sort, as

$$\forall x: \text{human} \; \exists y: \text{human} \; [\text{mother}(y, x)]\dagger$$

This reduction in complexity of axioms not only improves the readability of the knowledge base, but it also causes a significant decrease in the size of the search space for proofs. Cohn[22] (in the context of a resolution based theorem prover), and Davies[23] (in the context of a natural deduction system) show that this reduction can amount to as much as an order of magnitude.

Second, the sort hierarchy allows the knowledge engineer to represent what Clancey[13] calls 'structural knowledge', representing the taxonomical hierarchy of the application domain. In sorted logics, the unification algorithm has to take the sort hierarchy into account. The rules for unification in a sorted logic are:

a constant $c_1{:}t_1$ unifies with a constant $c_2{:}t_2$ if $c_1 = c_2$ and $t_1 = t_2$
a variable $x{:}t_1$ unifies with a constant $c{:}t_2$ if $t_2 \subset t_1$
a variable $x{:}t_1$ unifies with a variable $y{:}t_2$ if $t_1 \cap t_2 \neq \emptyset$

The third rule results in restricting the sorts of both variables to $t_3 = t_1 \cap t_2$. In order to guarantee that t_3 can be computed and is itself a legal sort, it is necessary that the system knows the values of all pairwise intersections of all sorts, and that sort-intersection is closed over sorts. In other words, if T is the set of all sorts (including $\emptyset$), then

$$\forall t_1, t_2{:} t_1 \in T \;\&\; t_2 \in T \rightarrow t_1 \cap t_2 \in T$$

Furthermore, in order to guarantee that t_3 is uniquely defined, we require that no two sorts are used to represent the same set of individuals. If this were allowed, the unification algorithm would no longer be able to return a unique most general unifier.

The knowledge engineer declares the sorts of the logic by declaration first the set of sorts, and then the hierarchy of sorts using set-theoretic primitives, such as equality, subset, superset, intersection, union, set-difference, complementation and disjointness. After this has been done, the system automatically checks whether the sort hierarchy satisfies the following criteria:

- No two sorts are equal.
- No sort except $\emptyset$ is empty.
- Intersection is closed over sorts.

If any of these constraints is violated, or if the constraints are not satisfied by the current declarations (because the knowledge engineer has underspecified the sort hierarchy), the system asks the knowledge engineer for different or additional declarations.

The sorts of the constants of the logical language can be declared in two ways. The simplest way is by simply enumerating the constants of a particular sort ('extensional' definition of types). However, for some sorts enumeration is not feasible, either because the set of constants of that sort is not known in advance, or because this set is too large, or indeed infinite. For these cases it is possible to define constants by declaring a 'recognition procedure' for the sort. Every object for which the recognition procedure succeeds will be assumed to be a constant of that particular sort ('intensional' definition of types). The definitions of the

*The empty sort, notation $\emptyset$, is the only sort representing an empty set of individuals, and as such does not fulfill any representational role. The reason why sorts must correspond to non-empty sets is discussed below.

†The notation $x{:}t$ is used to indicate that variable x is of sort t.

recognition procedures must be supplied in the implementation language of the system. This allows an interface between the logical representation language and the computational data types supplied by the implementation language of the system. An example of a sort that can be declared via this mechanism is the sort of all integers, or as a second example, for any given sort *T*, the sort *T List* consisting of all lists of elements of sort *T*.

As part of declaring the vocabulary of the logical representation language, it is possible to exploit what Weyhrauch[24] calls 'semantic attachment'. This is done by declaring a special class of predicates called 'evaluable predicates'. When one of these predicates is encountered in a proof, it is possible (depending on the control decisions made by the meta-level interpreter, discussed below) to execute a procedure defined for this predicate to determine its truth value, and possibly provide bindings for any variables. These predicates provide an interface between the logical representation language and the computational environment of the system, enabling external systems interaction, input/output for interacting with the end-user, and the access of the facilities provided by the implementation language of the system.

At this stage the representation language consists only of a sort hierarchy and a set of predicates, constants and functions. We still need to extend our language with 'logical connectives', such as implication, disjunction, etc., and possibly non-standard operators*, such as modal and temporal operators. As part of the declaration of these logical connectives the knowledge engineer can declare their properties with respect to commutativity and associativity. These declarations will be used to configure a unifier for the defined logical language (see the section below on the declaration of the proof theory for a more detailed discussion of this subject). Socrates distinguishes between different kinds of logical connectives: 'Unary' connectives take one argument; binary connectives take two arguments, and can be divided on the basis of their commutativity. Implication ($\rightarrow$) is an example of a 'non-commutative binary' connective, whereas equivalence ($\leftrightarrow$) is an example of a 'commutative binary' connective. If connectives are both commutative and associative (such as, for instance, conjunction), they are treated as so-called 'set-connectives'. This amounts to them taking any number of arguments in any order. Furthermore, set connectives are assumed to be idempotent: all multiple occurrences of an argument can be reduced to only one occurrence. Other possible connectives are not currently implemented in Socrates: 'bag' operators, which are associative and commutative but not idempotent (i.e. the order of arguments does not matter, but multiple occurrences of an argument cannot be reduced), and 'sequence' operators, where the order of the arguments is significant, and multiple occurrences cannot be reduced.

Storage and retrieval mechanism

Now that the vocabulary of the logical language is complete we need to declare a storage and retrieval mechanism for expressions in the language. This is what is typically called the 'knowledge base' of the system. The knowledge base of Socrates is not a flat space of assertions in the declared logical language, but can be divided into a number of 'partitions'. Each of these partitions can have a separate logical language associated with it (but see below for a restriction on this). This facility serves a number of different purposes.

First, it allows the knowledge engineer to use mixed language representations of the domain. Different types of knowledge or different aspects of the domain can be represented in separate languages. A particular example of this is described below, where we discuss the control knowledge of the system. This meta-level knowledge is expressed in a different language from the object-level knowledge. Nevertheless, it can be stored in the same knowledge base, using the partitioning mechanism to separate the two.

Second, the partition hierarchy can be used to reduce the search that needs to be done both by the retrieval mechanism and by the inference machinery. In many problem solving situations, only a subset of all the available knowledge is applicable at any one time to a given problem. Partitioning allows useful subsets to be applied while others are ignored. In this way, Socrates can be used to model a blackboard architecture, with each of the partitions simulating the contents of a knowledge source.

An important aspect of the partitions in the knowledge base is that they can be recursively divided into subpartitions, and that these subpartitions are organized hierarchically. Partitions lower down in this hierarchy inherit all the propositions from partitions higher up in the hierarchy, but not *vice versa*. A particular example of this could be the use of a single working memory for a number of different subsets of domain knowledge, where the working memory would be represented in the top node, and the subsets of domain knowledge represented in subpartitions. The results of reasoning done within one subpartition can in this way be communicated to the other partitions via the working memory. Again, this feature can also be used in the modelling of a blackboard system in Socrates, since the blackboard needs to be visible from all knowledge sources, but not *vice versa*. Since partitions can be created dynamically at runtime, another application for the partition hierarchy in the knowledge base is hypothetical reasoning, sometimes known as 'what-if' reasoning. Each time a new hypothesis is generated, a new subpartition can be created containing this hypothesis, and inheriting all existing propositions from higher partitions.

The inheritance mechanism puts restraints on the logical languages that can be used within subpartitions. Because subpartitions inherit propositions from superpartitions, all the partitions in a partition hierarchy must be associated with the same logical representation language. (Strictly speaking, it is only necessary that the language of a subpartition is an extension of the language of its superpartition, but Socrates does not support this type of language inheritance.) Thus, the knowledge base can be seen as a set of trees of partitions. Within each tree, all partitions must have the same language, but each separate tree can be associated with a different language.

A final facility supported by the knowledge base is

*The terms logical connective and logical operator are used synonymously

the annotation of propositions. Each proposition can be annotated with arbitrary information using a slot-value mechanism. This allows us to associate extra-logical information with propositions. This information can be used for a wide range of purposes, of which a few examples are: natural language descriptions to be used in explanations, control information to be used by the meta-level interpreter, certainty values, etc. This slot-value mechanism could be used to model a truth-maintenance system in Socrates, by storing each derived proposition in the knowledge base together with annotations that contain the premises that were used in its derivation.

Socrates uses a discrimination net technique for retrieval of propositions. The optimal criteria that the net should use for discrimination depend on the particular application, and Socrates therefore allows the knowledge engineer to adjust these discrimination criteria to suit the particular ways in which the knowledge base retrieval mechanism will be used. For example, if the main control regime for a particular application is some form of backward chaining, then propositions of the form 'left-hand-side → right-hand-side' will be retrieved from the knowledge base on the basis of the patterns in the 'right-hand-side' argument. This implies that the discrimination net should first discriminate implications on the basis of their consequent, rather than their antecedent. As a second example, we note that many predicates use a certain number of their arguments as 'inputs', while others are used as 'output' arguments. A predicate such as 'suffers-from (patient, disease)' will typically be used to associate a given patient with a disease, rather than to try and find all patients suffering from a given disease. This particular use of arguments indicates that the discrimination net should use the first argument as the main discrimination criterion, rather than the second argument. Because these properties are specific to a particular application, they can only be adjusted by the knowledge engineer who configures the system, rather than being hardwired into the retrieval mechanism of the knowledge base.

PROOF THEORY DECLARATION

The declaration of a logical language together with a storage and retrieval mechanism allows us to represent knowledge, but in order to manipulate this knowledge to derive new conclusions we need rules that tell us what the legal derivations will be. Since we have chosen logic as the representation language for the system, we are committed to logical deduction as the inference process. We therefore need to define a proof theory, i.e. a set of inference rules* that tell us how logical propositions can be manipulated in order to perform a proof. In Socrates we follow Bledsoe[25] in using the natural deduction style of performing proofs rather than, for instance, resolution. Although natural deduction systems use a relatively large number of inference rules (as opposed to the single inference rule of resolution based systems), and thereby create a potential control problem, a number of reasons can be given in favour of natural deduction. The inference rules of a natural deduction system are more intuitively meaningful than, for instance, the resolution rule. Furthermore, no normal forms are required for the formulas used in a proof. As a result, the proofs performed by a natural deduction system are easier to follow for a human reader, thereby improving the possibilities for explanation facilities. The naturalness of the proof development also makes it easier to identify heuristics to control the problem solving process, in the manner discussed below.

When expressing inference rules, the knowledge engineer can make use of extra-logical variables that range over well formed formulas from the object-level representation language. For instance, the rules

$$P, P \rightarrow Q \vdash Q$$
$$P \vdash P \vee Q$$

represents the rules for Modus Ponens and Disjunction Introduction. It is important to stress that these inference rules can be used in both a forward and a backward direction. For instance, Modus Ponens can be used to determine that P and $P \rightarrow Q$ have to be proved in order to prove Q (backward use), or the rule can be used to infer that Q is true when we know that both P and $P \rightarrow Q$ are true. Furthermore, because of the associativity and commutativity of some of the logical connectives, a single inference rule can often be applied in more than one way. For instance, if disjunction (V) has been declared as a 'set-operator', Disjunction Introduction can be applied backward to $(f \vee g)$ in two different ways, binding P to either f or g, thereby generating either f or g as a sub-goal for proving $(f \vee g)$. However, which of these possible applications of an inference rule should be used is a control decision, and is therefore a meta-level issue, which is not decided as part of the proof theory.

Not all the inference rules that the system uses are declared as part of the proof strategy. First of all, there is a set of inference rules that tell the system how to deal with typed quantification. These rules:

$\forall x{:}t_1\ P[x] \vdash P[c{:}t_2]$ for an arbitrary constant c, and all sorts t_1 and t_2 with $t_1 \supseteq t_2 \supset \emptyset$
$P[c{:}t_1] \vdash \exists x{:}t_2\ P[x]$ for an arbitrary constant c, and all sorts t_1 and t_2 with $t_2 \supseteq t_1 \supset \emptyset$*
$\exists x{:}t_1\ \forall y{:}t_2\ P[x,y] \vdash \forall y{:}t_2\ \exists x{:}t_1\ P[x,y]$ for all sorts t_1 and t_2

are taken to be of universal validity (that is: across different application areas of the system), and are therefore hardwired into the retrieval mechanism. A second set of rules, that is part of the retrieval mechanism in the knowledge base rather than the proof theory, are the results that deal with the commutativity and associativity of certain logical connectives. For any operator ϕ that has been declared as 'binary-commutative', the inference rule

$$P \phi Q \vdash Q \phi P$$

is hardwired into the knowledge base retrieval mechanism, as are, for every 'set-operator', the additional rules

$$P \vdash P \phi P$$
$$(P \phi (Q \phi R)) \vdash ((P \phi Q) \phi R).$$

*We use the term 'inference rule' in the logician's sense: an inference rule is a rule that describes how true formulas can be inferred from other true formulas, and is of the form $f_1 \ldots f_n \vdash g$, where g is inferred from $f_1 \ldots f_n$. The term 'inference rule' is often incorrectly used to denote formulas of the form $f \rightarrow g$ (i.e. logical implications).

*It is exactly because of this rule that in the section on logical representation declaration we required all sorts except $\emptyset$ to be non-empty.

By taking all these rules out of the explicit declaration of the proof strategy and transferring them to the retrieval mechanism of the knowledge base, we have given a limited deductive capability to the knowledge base.

The retrieval mechanism comprises two sequential stages, a syntactic 'pattern matcher' and a 'unifier'. Syntactic pattern matching is effected using the discrimination net technique mentioned above. Given a formula as a query to the knowledge base, this process will retrieve all formulas with the same pattern of operators and predicates, subject to the commutativity and associativity rules as declared for the particular language. This means that unification will be attempted only on patterns that have a strong possibility of succeeding.

The patterns that can be specified as input to the syntactic pattern maching phase are allowed to contain 'meta-logical (propositional) variables' that can match with formulas of the logical representation language rather than with terms. These meta-logical variables will be bound to the corresponding components of the matching expression, using pattern matching under the inference rules governing commutativity and associativity. For example, the pattern (*& f ?P*) will match with a formula like (*& g f*), binding *?P* with *g* after applying commutativity. To facilitate the retrieval of expressions containing set-operators, a special version of these meta-logical variables is available, the so-called 'segment-variables'. These segment variables do not match with formulas, but with lists of logical formulas. For example, a formula like (*& f g h*) will match with a pattern like (*& g ?P^segment*), binding *?P* with the list (*f h*). Because of the meta-logical variables, the patterns sent to the knowledge base for retrieval are actually schemata, representing whole families of queries rather than just a single query.

In the second phase of the retrieval process the unifier will construct bindings of the logical variables occurring in the query. This is necessary since in the context of expert systems we are interested in performing constructive proofs, and we therefore need the values for the existentially quantified variables in the query for which the query succeeds. In other words, for a query such as $\exists\, x{:}t_1 p(x)$, we want not only a yes/no response, but also the values of x which can be deduced from the contents of the knowledge base. Notice that these bindings might only consist of restrictions on the sort of x, rather than of bindings of x to terms. The sorted unification algorithm might tell us that the query succeeds for all values of x of a certain sort t_2, with $t_2 \subset t_1$. In this way, taxonomic reasoning is performed at retrieval time.

A final point to be made about the declaration of the proof theory concerns the soundness and completeness of the set of inference rules. In order to guarantee soundness of the proof theory, the knowledge engineer should not be allowed to declare arbitrary inference rules, but only to select inference rules from a predefined (and sound) set*. This selection process will of course affect the completeness of the system. However, the loss of completeness in the context of expert systems is not serious, since one does not want to infer *all* facts that follow from the available knowledge, but only those facts that one is interested in.

*Such a selection procedure has not been provided in the current implementation of Socrates.

PROOF STRATEGY DECLARATION

At this point, the knowledge engineer has declared both a logical language and a corresponding proof theory. From a purely logical point of view, no further declarations are necessary. The combination of language and proof theory determines all the possible inferences that the system can make. However, in order to create a practical computer system, one more step has to be taken. Proving statements in any non-trivial logical language is a search intensive problem. The logical language and proof theory together define a search space for the proof process. What remains to be done is the specification of the strategy that the system should use to traverse this space while searching for a proof. For this task, Socrates provides a declarative language for representing such a control strategy, described below. Because such a declarative language has certain disadvantages associated with it, a more procedural language has also been investigated (see below).

Declarative representation of control knowledge

Socrates allows the knowledge engineer to explicitly specify a control strategy. This control strategy provides the system with a description of its desired behaviour, and is interpreted at run time by the meta-level interpreter. As a result, the meta-level interpreter executes this control strategy, and thereby guides the search through the space of all possible proofs. The language that is used to express the control strategy is a many sorted version of Horn Clause Logic. This language, although also a logical representation language, should be distinguished from the logical languages used to represent the domain knowledge. Unlike the object-level languages, the language used at the meta-level has a fixed set of logical connectives, namely exactly those connectives needed in Horn Clause Logic: conjunction, implication and negation, plus disjunction. All these connectives are declared as non-commutative, non-associative. This is done because the procedural interpretation (i.e. the way in which the meta-level interpreter executes expressions of the language) is also fixed. The procedural interpretation of the language is the standard interpretation for Horn Clauses, the standard depth-first proof procedure as found in Prolog systems. The reason why Socrates does not allow the knowledge engineer to change the control regime of the meta-level interpreter (which would amount to providing a meta-meta-level interpreter†) follows from the analysis of models of rationality, sketched above. As indicated, typical expert system tasks such as diagnosis, planning, monitoring, etc. are related to particular control regimes. The meta-level controls the behaviour of the object-level interpreter according to the expert system task. The variation in control is achieved by changing the meta-level knowledge base. There is no need to change the interpreter, which always has the same task, namely controlling the behaviour of the object-level interpreter by using the data in the meta-level knowledge base.

The parts of the meta-level language that are still subject to declarations made by the knowledge engineer

†The notion of a meta-meta-level interpreter should not be confused with the scheduler; the relation between meta-level interpreter and scheduler is very different from the relation between object-level and meta-level interpreter.

are therefore the set of constants, predicates and function symbols, the sort hierarchy, and the set of 'evaluable predicates'. Figure 2 shows an example of a description of a local-best-first non-exhaustive backward chaining control strategy. For this control strategy a sort hierarchy was defined containing the sorts formula, list-of-formulas, substitution and list-of-substitutions. The sort formula was further subdivided into compound-, evaluable- and non-evaluable-formula. A further specialization of list-of-formulas was non-empty-list-of-formulas. In the example shown in Figure 2, clause (1) states that in order to prove a non-compound expression F on the basis of the contents of knowledge base partition P giving a substitution S as a result, the system should either try to see if the formula is a known fact in the knowledge base, try to infer the formula on its own, or it should ask the end-user. Trying to infer the formula means generating all possible inferences, selecting some of these possible inferences, and continuing with clause (2). That clause chooses the best of all selected possible steps, and tries to continue the proof with this selection. If this succeeds, the proof terminates (i.e. non-exhaustive); if this fails, the proof continues with the next best step. Clause (3) states the criterion used in the best-first search, while clause (4) describes what needs to be done in order to prove a compound expression: prove both left- and right-hand sides of the compound expression, and combine the results. Clause (5) states that all evaluable predicates encountered in a proof should be evaluated without any further control scheduling.

This example shows how the different aspects of this strategy can be changed if needed for a particular application. For example, the order of the disjuncts in clause (1) might be changed to ask the end-user for solutions before the system tries a proof itself, or the ask-user disjunct might be deleted altogether. The criterion used for the best-first scheduling could be changed, or a new decision for scheduling the order in which conjuncts are proved in clause (4) could be introduced. More thorough changes to the strategy could also be made, but they would amount to writing a completely new proof strategy rather than changing the one shown in this example.

```
(1) (∀ F:non-evaluable-formula, P:partition, S:substitution,
    Next:non-empty-list-of-formulas,
    SomeNext:non-empty-list-of-formulas)
    [knowledge-base-lookup(F, P, S)
      ∨ ( object-level-interpreter(F, P, backward, Next) &
        select-inferences(Next, SomeNext) &
        infer(SomeNext, P, S))
      ∨ ask-user(F, P, S))
      → proof(F, P, S)]

(2) (∀ Inferences:non-empty-list-of-formulas, P:partition,
    S:substitution, Best:formula, Rest:list-of-formulas)
    [best(Inferences, Best,Rest) & (proof(Best, P, S) ∨
    infer(Rest, P, S))
    → infer(Inferences, P, S)]

(3) (∀ List-non-empty-list-of-formulas,
    Best:formula, Rest:list-of-formulas)
    [higher-certainty-value(List, Best,Rest)
    → best(List, Best,Rest)]

(4) (∀ F:compound-formula, P:partition, S:substitution,
    Lhs:formula, Rhs:formula, LhsSubst:substitution,
    RhsSubst:substitution)
    [split-compound-expression(F, Lhs, Rhs) &
    proof(Lhs, P, LhsSubst) & proof(Rhs, P, RhsSubst)
    &-combine(LhsSubst,
    RhsSubst, S) → proof(F, P, S)]

(5) ∀ F:evaluable-formula, P:partition, S:substitution.
    [evaluate(F, P, S) → proof(F, P, S)]
```

Figure 2. Declarative specification of control in Socrates

Of the evaluable predicates in Figure 2 (such as ask-user, knowledge-base-lookup, combine) the predicate object-level-interpreter is the most important one. This predicate encapsulates the interface between the meta-level interpreter executing the control strategy, and the object-level interpreter handling the logical representation language and the corresponding proof theory. The input of this predicate is an object-level formula F, the name of a knowledge base partition P and a direction in which to apply inference rules (either forward or backward), and returns as output the result of applying all inference rules specified as part of the proof theory for partition P to the input formula F in the indicated direction. In terms of the proof search space, this amounts to generating all nodes that are accessible from the current node as represented by F. Thus, the predicate object-level-interpreter allows the meta-level interpreter to access an explicit representation of the object-level search space, and to choose which branches of the object-level proof tree will be expanded on the basis of the control regime provided by the knowledge engineer.

Unlike many logic-based meta-level architectures proposed in the literature, (such as Silver[19], or the Prolog system described in[26]), Socrates completely separates the languages used at the object-level from the language used at the meta-level. Even when the object-level representation language happened to be defined as sorted Horn Clause Logic, the two languages would still be syntactically separate. The meta-level and object-level languages are connected through a 'naming relation'. The meta-level language contains names for all object-level expressions. In Socrates, the name of an object-level sentence corresponds to a constant in the meta-level language. Other meta-level constants are used to denote bindings for object-level variables. If this is required, meta-level expressions could range over any of the extra-logical properties of object-level expressions, such as truth values, certainty factors, justifications, etc. In this way, for instance, Socrates could be configured to deal with certainty values by specifying as part of the control strategy how certainty values should be used in a proof. This corresponds to the approach suggested by Shapiro[27], with the important difference that Socrates makes a correct distinction between meta-level and object-level languages, whereas Shapiro confuses the two and uses Prolog for both.

A number of reasons can be given why it is important for the meta-level language to be separated from the object-level languages. First, there is an epistemological reason: as argued in[29], different domains require different representation languages, and the object-level and the meta-level of Socrates deal with widely different domains (the object-level deals with the application domain of the system, while the meta-level deals with the issue of controlling the object-level). A second argument concerns the modularity of the system: it should

be possible to vary control knowledge and domain knowledge independently. The third argument is one of explanation: in order to enable the system to include control knowledge explicitly in its explanations, it is important for both the human reader and the automated explanation generator that control knowledge can be syntactically distinguished from domain knowledge.

Procedural representation of control knowledge

The approach to the specification of a proof strategy described above is based on the use of a declarative meta-level language. Although the declarative style of control of reasoning has its attractions, there are also disadvantages. Two problems in particular are caused by the use of a declarative language, and in an attempt to overcome these problems Socrates provides an alternative, more procedural language for specifying control regimes. First, the extra layer of interpretation that is incurred by the explicit meta-level interpreter is expensive, because of the declarative nature of the meta-level language. A procedural meta-level language would be much closer to the underlying implementation language and machine architecture, and therefore cheaper to execute. Second, much of the knowledge expressed in the meta-level language is procedural, rather than declarative. For example, one often wishes to apply knowledge of the form 'try method-1 before method-2', or 'in order to achieve goal-1, achieve sub-goal-1 to subgoal-n'. In the declarative meta-level language this type of procedural knowledge has to be expressed by either relying on the hardwired control regime for the meta-level interpreter, or by using semantic attachment. Neither of these ways of expressing procedural knowledge is very desirable, since they encode knowledge implicitly rather than represent it explicitly. A more procedural language provides a more natural medium for expressing the procedural control knowledge.

Our approach to procedural control is therefore to provide a 'meta-level programming language' rather in the vein of ML[30] in its relationship to LCF. That is, we provide high level primitives that make the writing of control regimes easier. It is important to note that a procedural control language is only an alternative way of implementing the separation of control knowledge from object-level knowledge. The procedural approach still clearly separates control knowledge from object-level knowledge. The difference is that rather than putting the control knowledge into a declarative knowledge base with its own interpreter, we propose implementing a special purpose meta-level interpreter that incorporates the control knowledge. The three stage process of building systems is retained. One particular point to note is that we retain the explicit declaration of inference rules and provide primitives to apply inference rules. The procedural meta-level language consists of the implementation language of the system (Common Lisp), extended with primitives implementing standard artificial intelligence techniques that have been found useful in writing interpreters.

Central to the system is the use of lazy evaluation. The procedural meta-level language provides facilities for the manipulation of lazily evaluated lists, which form the basis for backtracking and coroutining in the control regimes. The second important component is agenda-based reasoning. This technique allows the knowledge engineer to experiment with several different control strategies, often only changing the way the agenda is handled without changing the rest of the control regime code. The third component is a pattern matcher which provides the basis of pattern directed invocation of proof methods.

```
(defun proof (GoalList Partition Subst &aux NewGoals
NewSubst)
  (if GoalList
    (foreach (NewGoals . New Subst)
      in (or (any-of (order-inferences (evaluate (first
      GoalList)
              :partition Partition :subst Subst))
          (order-inferences (lookup (first GoalList)
              :partition Partition :subst Subst))
            (select-inferences
              (generate-backward-inferences
              (first GoalList)
                :partition Partition :subst Subst)))
          (ask-user (first GoalList) :partition Partition
              :subst Subst))
      generate-each (proof (append NewGoals
      (cdr GoalList))
              Partition NewSubst)) (list Subst)))
```

Figure 3. Procedural specification of control in Socrates

It is important to realize that in this procedural approach it is still the case that the only inference rules are those declared explicitly during the declaration of the proof theory (as described above). Language primitives are provided to apply the declared inference rules.

Figure 3 shows an example of a control regime formulated in the procedural meta-level language. This code is the procedural equivalent of the declarative prover given in Figure 2. The main function is called 'proof' and performs the same function as the predicate of that name. Being stream-based, proof returns a stream of substitutions that prove the goals in the 'GoalList' argument. Thus, if the GoalList is empty there is one such proof, given by the Subst argument. If the GoalList is not empty then the inference rules are applied to the first goal in GoalList. The 'any-of' macro is a way of lazily combining streams. Thus, in this example, we generate all the possible inferences using 'evaluate' (and put them in a preferred order), then all the possible inferences using 'lookup' (again in preferred order), finally followed by a selection of the possible inferences generated using all the inference rules. 'If and only if' no possible inferences are generated by this procedure, then 'ask-user' is applied to obtain possible inferences. Each inference rule application generates three things: a set of new goals to be proved in order to prove the goal (bound to the variable 'NewGoals'); a new substitution, bound to the variable 'NewSubst'; and a justification, which merely describes which inference rule has been applied and is effectively ignored by this prover. The 'foreach' macro itself generates a stream of answers. Thus, if at some later stage in the proof there is a need to backtrack, the next inferences will be generated only then. Note that, unlike the declarative prover, which generates only the first proof, the procedure 'proof' returns a stream of proofs, and thus generates the set of all proofs (lazily).

It was noted at the beginning of this section that much

control knowledge seems very procedural in nature (such as the execution of a sequence of goals). However, sometimes control knowledge is declarative in nature (e.g. a set of criteria used in the ordering of conjuncts). The architecture of Socrates is such that even with procedural control it is still possible to invoke a declarative meta-level interpreter that is implemented using the techniques described above.

Although the above features provide a basis for a procedural language for formulating control regimes, certain problems remain. First, the current language may not be powerful enough, and further additions may be needed. Second, although in one sense the current procedural meta-level language might not be powerful enough, in another sense it might be too powerful. As described, the procedural meta-level language consists of extensions to Common Lisp, thereby making all the general purpose expressive power of Lisp available to the knowledge engineer when writing control regimes. It might well be the case that this provides too powerful a language, since it does not restrict the knowledge engineer in any way.

The scheduler

A third level in the architecture of Socrates (see Figure 1) is the 'scheduler'. This third level is not actually implemented in the current Socrates architecture, but it can be added to the current system with little effort. The main notion that is treated at this level is that of a 'subtask'. As shown in Reichgelt and van Harmelen[29], many expert systems perform not just one simple task, but a composite one that can be thought of as consisting of a number of elementary tasks (MYCIN, R1 and VM are among the systems discussed in that paper). It is unlikely that one appropriate control regime can be found that would be suitable for these composite tasks. Rather, the composite task should be split up into its constituent subtasks, and a proper control regime can then be chosen for each of the subtasks.

The subtasks that would result from this decomposition process are the kind of prototypical tasks proposed in Reichgelt and van Harmelen[29], Chandrasekaran[6], and Breuker and Wielinga[5], like classification, monitoring, simulation, design, etc. The scheduling level of the Socrates architecture is meant to deal with this subdivision of the major task into prototypical subtasks. Each of these prototypical subtasks can then be solved using the appropriate meta-level control strategy. (By using the knowledge base partition mechanism, it is possible to equip a Socrates configuration with more than one control strategy.) For engineering purposes it would be easiest to equip the scheduling level with a language similar to (but again syntactically separate from) the language used to describe the control strategy at the meta-level. However, early experience indicates that the type of knowledge to be expressed at the scheduling level is of a very procedural nature (even more so than the knowledge expressed at the meta-level), and therefore a language with more conventional procedural primitives, such as sequences, conditionals, loops and subroutines, might be more appropriate.

PRACTICAL PROGRESS AND ACHIEVEMENTS

A version of the Socrates architecture as described above has been implemented in approximately 15K lines of Common Lisp code. (Notice the distinction between 'implementation language' and 'representation language' of a system: although the knowledge representation and inference process of Socrates are both logic based, the system is *not* implemented in Prolog or any other logic programming language.) The system currently runs in a number of Common Lisp implementations on UNIX based systems. In one of these Common Lisp systems, the Poplog system, a graphics-based knowledge engineer interface, has been constructed, allowing interaction with the system via menus, browsers, graphers, etc.

A substantial set of different control strategies has been written as meta-level programs, including backward and forward chaining, exhaustive and non-exhaustive search, user guided or automatic conflict resolution, best-first, depth-first, breadth-first search, branch and bound type algorithms, generate and test procedures, elimination and confirmation strategies, etc.

A number of demonstration systems have been built using Socrates, including an expert system in the domain of personal investment advice and a route planning system. More significantly, Socrates has been used to reimplement an existing expert system under the name of DOCS, developed by GEC Research in collaboration with Westminster Hospital, London. This system consists of 130 rules divided over two knowledge base partitions. The partition hierarchy is organized so that these two partitions can use data from a common working memory partition. A sort hierarchy of over 80 sorts was used to model the taxonomic hierarchy of the medical domain. A generate and test control strategy was specified for this system, consisting of some 20 clauses in the meta-level knowledge base.

In the area of theorem proving Socrates has been used to solve the problem described by Walther[21], known as 'Schubert's steamroller'. This problem was originally formulated because of the huge search space that it generates. Using the sort hierarchy to model the taxonomic part of the problem, a breadth-first search strategy, implemented using our procedural control techniques, resulted in the first natural deduction style proof for this problem. A number of other procedural control strategies were implemented to solve the problem, and a comparison of these different strategies showed the importance of the exploitation of meta-knowledge to guide the search for a proof. This application of Socrates is described in detail by Davies[23]. Current areas of activity are:

- The use of meta-level interpretation for dealing with extra-logical issues, such as uncertainty, truth maintenance, explanation, etc. (how the slot-value annotation mechanism of Socrates' knowledge base opens up these possibilities is described above).
- The implementation of modal logics through the use of reification, as described by Reichgelt[31].
- The classification of domains and subtasks, as stated by Reichgelt and van Harmelen[28,29], in order to provide the knowledge engineer with guidelines that indicate how to choose the appropriate representation language given a particular application domain, and how to choose the appropriate control regime given a particular prototypical task.
- The development of a library of control regimes that

can be executed by the system. Rather than having to write a new control strategy from scratch every time, the knowledge engineer can use a library of preprogrammed control strategies. The knowledge engineer can then either use one of the strategies directly from the library, or use one of the library elements as the basis for his own control strategy by making small changes to the preprogrammed strategy.

OPEN PROBLEMS

Of the three stages of the configuration process, the third one (declaring a proof strategy) is by far the most problematic. An important open question here is the choice of a good language for specifying such a control strategy. As described above, the system primarily uses a declarative logical language to do this, but a procedural language has also been investigated. Apart from the type of language used at the meta-level, a related problem is the required vocabulary of such a language. At the moment, the vocabulary of the system is specified by the knowledge engineer, and although this is to a certain extent inevitable, since part of the vocabulary will be application-specific, one would hope that at least a central core vocabulary can be distinguished that can be preprogrammed into the system. The example of a control regime discussed above suggests predicates to do with manipulating substitutions and formulas, and with generating the object-level search space, but a more extensive and more exactly defined vocabulary is needed in order to alleviate the task of the knowledge engineer.

A second problem associated with the explicit meta-level interpreter is that of meta-level overhead. Although the flexibility in defining the appropriate control strategy at the meta-level can considerably reduce the object-level search space, the price we have to pay for this is the fact that the object-level inference process is completely simulated by the meta-level interpreter. This is obviously much more expensive than an object-level interpreter that has the appropriate control strategy hardwired into it. This problem could be solved by taking the explicit formulation of a control regime, and compiling it into an interpreter that has the particular control regime hardwired into it. This compilation process (whose first stages could be similar to that described by Altman and Buchanan[32]) has been simulated in Socrates by hand coding a number of hardwired control strategies. Experience with these hardwired strategies in both the DOCS system and in solving Schubert's steamroller indicates that the meta-level overhead can indeed be reduced to an acceptably small amount.

CONCLUSION

The work described in this paper attempts to create an environment for building expert systems based on the following principles:

- An epistemological analysis of the domain and task of a particular application guides the choice of the appropriate knowledge representation language and the appropriate control regime.
- Logic is used as the main underlying formalism.
- Control knowledge is represented explicitly and is separated from the domain knowledge.

When configuring the Socrates environment into a particular expert system, a knowledge engineer can vary the architecture along three dimensions:

- The representation language: a knowledge engineer can define his own logical representation language, including first order logics (possibly many-sorted), modal logics, temporal logics, etc.
- The inference rules for the logical language: the set of rules that determine the possible inferences made in the logical language can be changed by the knowledge engineer.
- The control regime under which the inference rules will be used to perform proofs in the logical representation language.

An implementation of the Socrates abstract architecture and a number of applications of the system have proved the feasibility of this approach.

Due to limitations of space, some of the arguments and descriptions in this paper are rather terse. A long version of this paper can be found in Corlett, Davies, Khan, Reichgelt and van Harmelen[7].

ACKNOWLEDGEMENTS

As the grant holder, Peter Jackson has made many contributions to the work described above. Ray McDowell and Dave Brunswick have contributed to the work done at GEC Research. The research was carried out as part of Alvey Project IKBS/031 in which GEC Research, the University of Edinburgh and GEC Avionics were partners. The university work was supported by SERC grant GR/D/17151.

REFERENCES

1 **Alvey, P** 'Problems of designing a medical expert system' *3rd Tech. Conf. British Comput. Soc. Specialist Group on Expert Syst.* (December 1983) pp 20–42
2 **Bobrow, D and Stefik, M** *The LOOPS Manual* Rank Xerox, UK (1983)
3 *The knowledge engineering environment* Intellicorps, California, USA (1984)
4 **Williams, C** *ART, the advanced reasoning tool, conceptual overview* Inference Corporation, Los Angeles, California, USA (1983)
5 **Breuker, J A and Wielinga, B J** 'Models of Expertise' *Proc. 7th Europ. Conf. on Artif. Intel.* (July 1986) pp 306–318
6 **Chandrasekaran, B** 'Towards a functional architecture for intelligence based on generic information processing tasks' *7th Int. Joint Conf. on Artif. Intel.* (August 1987) pp 1183–1192
7 **Corlett, R, Davies, N, Khan, R, Reichgelt, H and van Harmelen, F** 'Socrates: a flexible toolkit for building logic based expert systems' in **Jackson, P, Reichgelt, H and van Harmelen, F (eds)** *Logic-based knowledge representation* MIT Press, USA (1988)
8 **Davis, R** 'Meta rules: reasoning about control' *Artif. Intel.* Vol 15 No 2 (1980) pp 179–222
9 **Bundy, A and Welham, B** 'Using meta-level inference for selective application of multiple rewrite rules in algebraic manipulation' *Artif. Intel.* Vol 16 No 2 (1981) pp 189–212

10 **Clancey, W** 'The advantages of abstract control knowledge in expert system design' *Proc. 3rd Annual Meeting of the American Assoc. for Artif. Intel.* (1983) pp 74–78

11 **Aiello, L and Levi, G** 'The uses of metaknowledge in AI systems' *Proc. European Conf. on Artif. Intel.* (September 1984) pp 707–717

12 **Clancey, W** 'Representing control knowledge as abstract tasks and metarules' in **Coombs, M and Bolc, L (eds)** Springer-Verlag, FRG (1985). Also: Stanford Knowledge Systems Laboratory, Working Paper No KSL-85-16 (April 1985)

13 **Clancey, W** 'The epistemology of a rule-based expert system: a framework for explanation' *Artif. Intel.* Vol 20 (1983) pp 215–251. Also: Stanford Heuristic Programming Project, Memo HPP-81-17, STAN-CS-81-896 (November 1981)

14 **Bundy, A, Byrd, L, Luger, G, Mellish, C, Milne, R and Palmer, M** 'Solving mechanics problems using meta-level inference' *Proc. Int. Joint Conf. on Artif. Intel.* (August 1979). Also in **Michie, D (ed.)** *Expert Systems in the Micro Electronic Age* Edinburgh University Press, UK (1979) pp 50 64

15 **Davis, R** 'TEIRESIAS: applications of meta-level knowledge' in **Davis, R and Lenat, D B (eds)** *Knowledge-Based Systems in Artificial Intelligence* McGraw-Hill, New York, USA (1982) pp 227–490

16 **Pereira, L M** 'Logic control with logic' *Proc. 1st Int. Logic Programming Conf.* (1982) pp 9–18. Also in: **Campbell, J A (ed)** *Implementations of Prolog* Ellis Horwood, UK (1984)

17 **Genesereth, M and Smith, D** 'An overview of meta-level architecture' *Proc. 3rd Annual Meeting of the American Assoc. for Artif. Intel.* (1983) pp 119–124

18 **Sterling, L** 'Implementing problem solving strategies using the meta-level' DAI Research Paper No 209, Department of Artificial Intelligence, University of Edinburgh, UK (1984)

19 **Silver, B** *Meta-level Inference* Studies in Computer Science and Artificial Intelligence, North Holland, Amsterdam, The Netherlands (1986)

20 **Welham, B** 'Declaratively programmable interpreters and meta-level inferences' Hewlett-Packard Laboratories Bristol Research Centre Technical Memo No HPL-BRC-TM-86-027 (September 1986). Also in: **Maes, P and Nardi, D (eds)** *Meta-level architectures and reflection* North Holland, Amsterdam, The Netherlands (1987)

21 **Walther, C** 'A mechanical solution of Schubert's steamroller by many-sorted resolution' *Proc. 4th Annual Meeting of the American Assoc. Artif. Intel.* (1984) pp 330–334

22 **Cohn, A G** 'On the solution of Schubert's steamroller in many sorted logic' *Proc. 9th Int. Joint Conf. on Artif. Intel.* (August 1985) pp 345–352

23 **Davies, N** 'Schubert's steamroller in a natural deduction theorem prover' *Proc. 7th Tech. Conf. of the British Comput. Soc. Specialist Group on Expert Syst.* (December 1987)

24 **Weyhrauch, R** 'Prolegomena to a theory of mechanised formal reasoning' *Artif. Intel.* Vol 13 No 1 (1980) pp 133–170

25 **Bledsoe, W** 'Non-resolution theorem proving' *Artif. Intel.* Vol 9 No 1 (1977) pp 1–35

26 **Gallaire, M and Lasserre, C** 'Meta-level control for logic programs' in **Clark, K and Tarnlund, S (eds)** *Logic Programming* Academic Press, UK (1982) pp 173–188

27 **Shapiro, E** 'Logic programs with uncertainties: a tool for implementing rule-based systems' *Proc. 8th Int. Joint Conf. on Artif. Intell.* (August 1983) pp 529–532

28 **Reichgelt, H and van Harmelen, F** 'Relevant criteria for choosing an inference engine in expert systems' *Proc. 5th Tech. Conf. of the British Comput. Soc. Specialist Group on Expert Syst.* (December 1985) pp 21–30

29 **Reichgelt, H and van Harmelen, F** 'Criteria for choosing representation languages and control regimes for expert systems' *The Knowledge Eng. Rev.* Vol 1 No 4 (December 1986) pp 2–17

30 **Gordon, M, Milner, R and Wadsworth, C** *Edinburgh LCF: a mechanized logic of computation* Springer-Verlag Lecture Notes in Computer Science, Vol 78 Springer-Verlag, FRG (1979)

31 **Reichgelt, H** 'Semantics for a reified temporal logic' *Proc. 1987 AISB Conf.* (April 1987) pp 49–62

32 **Altman, R B and Buchanan, B G** 'Partial compilation of strategic knowledge' *Proc. 6th Annual Meeting of the American Assoc. for Artif. Intel.* (July 1987) pp 399–404