ELSEVIER

Artificial Intelligence 85 (1996) 3-44

Artificial Intelligence

A systematic methodology for cognitive modelling

R. Cooper[a,*], J. Fox[b], J. Farringdon[c], T. Shallice[c]

[a] *Department of Psychology, Birkbeck College, Malet Street, London, UK WC 1E 7HX*
[b] *Advanced Computation Laboratory, Imperial Cancer Research Fund, London, UK*
[c] *Department of Psychology, University College London, London, UK*

Received January 1995; revised August 1995

Abstract

The development and testing of computational models of cognition is typically *ad hoc*: few generally agreed methodological principles guide the process. Consequently computational models often conflate empirically justified mechanisms with pragmatic implementation details, and essential theoretical aspects of theories are frequently hard to identify. We argue that attempts to construct cognitive theories would be considerably assisted by the availability of appropriate languages for specifying cognitive models. Such languages should: (1) be syntactically clear and succinct; (2) be operationally well defined; (3) be executable; and (4) explicitly support the division between theory and implementation detail. In support of our arguments we introduce Sceptic, an executable specification language which goes some way towards satisfying these requirements. Sceptic has been successfully used to implement a number of cognitive models including Soar, and details of the Sceptic specification of Soar are included in a technical appendix. The simplicity of Sceptic Soar permits the essentials of the underlying cognitive theory to be seen, and aids investigation of alternative theoretical assumptions. We demonstrate this by reporting three computational experiments involving modifications to the functioning of working memory within Soar. Although our focus is on Soar, the thrust of the work is more concerned with general methodological issues in cognitive modelling.

1. Introduction

The natural complexity and variability of ordinary human behaviour creates a need for considerable methodological sophistication in cognitive research. Consequently, students of experimental cognitive psychology are routinely trained in rigorous experimental design and data analysis. Notwithstanding the many ambiguities of cognitive theory, the

* Corresponding author. E-mail: r.cooper@psyc.bbk.ac.uk.

SSDI 0004-3702(95)00112-3

criteria by which experimental research are judged are public and largely uncontroversial. On the theoretical side the situation is less satisfactory. Most theoretical discussion in psychology is informal. Theories are often presented in terms of box and arrow diagrams (see, e.g., [3,29,42]), or draw on natural language metaphors (see, e.g., [41]), but the box-arrow notation and its underlying assumptions are generally poorly specified, and it has been argued that "attractive metaphors raise more obscurities than they resolve" [39, p. 402]. This informal approach restricts the field's ability to make clear theoretical statements.

In response, many cognitive scientists have looked for more satisfactory ways of formulating their theories. Frequently they have turned to computational modelling techniques (both connectionist and symbolic) for developing and presenting their proposals. Such techniques lead to precise statements of theory by forcing the detailed specification of all aspects essential to the theory's implementation, including many aspects which might otherwise be overlooked. Computational techniques also allow predictions to be drawn from those theories by simulation.

Sound methodological principles are necessary, however, if the full potential of computational techniques is to be delivered to cognitive science. Without such principles there are two clear dangers. Firstly, unconstrained programming allows the exploration of alternative computational mechanisms with little theoretical foundation. Such experimental programming may lead to programs which replicate empirical data, but the strategy is in direct conflict with the standard scientific method of evaluating a theory by testing its predictions. In addition, if such programs are to further our understanding of the phenomena in question, some form of reverse engineering is required in order to derive from the program an explanation of its behaviour. Secondly, the unconstrained implementation of a theory can lead to situations where the program details obscure the asserted theory. The abstract nature of psychological theorising requires that, for computational completeness, any computational instantiation of a theory will include aspects which are not theoretically motivated. Without appropriate methodological principles, the distinction within the implementation between theoretically motivated aspects and implementation details cannot be ensured. Moreover, the independence of simulation results from supposedly irrelevant implementation details cannot be determined.

The utility of sound, systematic methodologies can be seen in a number of fields. Computer science, for example, has developed rapidly in recent years, with the introduction of systematic software development methodologies (e.g., [37,47]) and formal languages for specifying program designs (e.g., [44]). AI technologists (those primarily interested in AI as a technical discipline rather than as a natural science) have, furthermore, successfully applied formal frameworks in many areas, such as computational linguistics (e.g., [5,19]), reasoning (e.g., [15,21]), knowledge representation (e.g., [12]), uncertainty and decision making (e.g., [1]) and expert systems (e.g., [45]). There is now a general consensus that increased use of formal design and systematic programming methods yields deeper understanding and greater predictability of software, leads to clearer communication, and facilitates independent assessment of programs and their underlying principles. While the methods and tools of mathematicians and engineers are not necessarily central to the natural sciences, whose core is empirical investigation, with better techniques for formulating and presenting computational theories, and a stronger methodology for developing and testing them, cognitive science would seem likely to advance more certainly.

Further motivation for the development of improved modelling methodologies comes from an examination of the goals and methods currently advocated in computational psychology. In particular, Newell has argued that "it is time to get going on producing unified theories of cognition—before the [empirical] database doubles again and the number of [theoretical] clashes increases by the square or the cube" [31, p. 25] and Soar [23,31,40] can be seen as the culmination of his long-argued position that psychological theorists must aim for cumulative development and avoid paradigm-related, transitory fashions [30]. In this vein Newell [31] argues that the work of the Soar community represents a research programme in the tradition of Imre Lakatos [25]. In such a programme any one incarnation of a theory may prove to be incorrect in peripheral details but advances in understanding are made possible by building progressively upon a hard core. In the case of Soar this progression is embodied in a series of related computer programs, each aiming to provide a more adequate theoretical account than its predecessors.

At this point, however, we have to question whether currently available methodologies for designing, implementing and documenting computational theories match up to Newell's challenge. Take Soar itself. From the point of view of theoretical methodology Soar seems more sophisticated than much current cognitive psychological modelling. Statements of Soar as a theory of cognition are, like most psychological theories, informal; a set of claims and assumptions about cognitive processing expressed in ordinary English (e.g., [31,40]). As we argue below, however, there is a substantial gap between these informal statements and the computational realisation of the theory as a LISP or C program. This gap has important consequences for evaluating Soar. It brings into question the degree to which Soar can truly be said to be a well-articulated cognitive theory, and the degree to which the implementations can be said to be accurate realisations of that theory (see [10]). This is not a criticism of Soar or the Soar community, but a general methodological weakness which undermines computational research. We need to be able to express our theories in ways that are more open to detailed scrutiny; less ambiguously enmeshed in particular technologies; more easily evaluated against generally accepted criteria of adequacy; and hence more easily compared.

This paper describes an attempt to address these problems by applying some lessons from AI and computer science. Our goal is an executable specification language: a language for expressing theoretical proposals which avoids, as far as possible, the difficulties mentioned earlier. Such a language should provide a notation for succinctly and clearly formulating proposals regarding the properties and interdependencies of putative cognitive mechanisms. In addition it should provide a way of executing or animating the theory to determine how it will truly behave. Finally, the language should explicitly support the distinction between theoretically motivated aspects and implementation details.

We begin with what is essentially a requirements analysis. We survey existing forms of theory articulation, highlighting the failings of a variety of current methods, before arguing for the utility of executable specification languages and the necessity of the theory/implementation distinction. In Section 3 we introduce Sceptic, an executable specification language which we have used to implement a number of current psychological theories, including Soar, and discuss the extent to which it addresses the issues detailed in Section 2. Section 4 relates our experience with Sceptic as a cognitive modelling language. Specifically, we consider the use of Sceptic in the large-scale reconstruction of Soar, the generality of Sceptic (as evidenced by the implementation of a number of less extensive theories), and the utility of Sceptic in facilitating computational experimentation (as evidenced by three experimental modifications to the Soar reconstruction). We conclude with a discussion of the role of a modelling language such as Sceptic in a wider methodology for cognitive modelling. For completeness, annotated details of the Sceptic specification of Soar are given in an appendix. While the work reported here is developed in the context of symbolic modelling, many of the methodological points apply equally well to connectionist modelling.

2. Issues in theory specification

Central to any systematic methodology for cognitive modelling must be techniques for the description or specification of theories. Within cognitive psychology, a number of such techniques are currently used, including natural language descriptions, information processing diagrams, computer programs, and (more rarely) formal specifications. Each of these approaches has its advantages, but each also has serious problems. In this section we examine the above forms of theory articulation and consider how their failings might be addressed.

2.1. Current forms of theory articulation: a critique

The relative merits of the various approaches to theory specification are most clearly illustrated with respect to a concrete example. In the following discussion we therefore focus on the use of these techniques in the description of Soar [23,31,40]. Soar is appropriate for our discussion because it has been described in natural language, as a computer program in procedural terms (LISP and C programs exist for various versions of the theory), and in the formal specification language Z [27]. However, whilst we concentrate in this section on the current forms of articulation of Soar, our comments apply equally to the methods used in the description of most other relatively well-developed cognitive theories (e.g., ACT* [2], Mental Models [20], etc.).

Soar is an information processing mechanism which is specialised to adaptively raise and resolve goals by exploiting a body of knowledge. The Soar architecture is based upon the well-established approach to modelling problem solving as search in a problem space [34]. Soar implements its search by cyclically retrieving knowledge relevant to its current goal (the elaboration phase), and making decisions about the results (the decision phase). Speaking loosely, decisions are made about the knowledge to use in resolving goals, the interpretations to assign to data, and the actions to carry out. If Soar has sufficient knowledge in a particular context the goal will be solved directly. If its knowledge does not embody a solution it will automatically generate a new goal to resolve the impasse and recursively attempt to solve that. If the subgoal succeeds, the impasse is resolved and the result becomes available in the parent goal context. While problem solving within a subgoal the architecture is able to identify critical contextual data and learn this as new knowledge for future use.

2.1.1. Natural language and diagrammatic descriptions

There are many natural language descriptions of Soar in the literature (see, e.g., [23, 31, 40]), often accompanied by diagrams. Such descriptions are generally abstract. They tend to present Soar as an information processing psychological theory, and in so doing employ the language and techniques of that field. The advantage of such descriptions is that they allow the theory to be stated at an appropriate level of abstraction and do not force theorists into describing unnecessary details. Soar, for example, embodies a set of preference rules for making certain choices during decision making; we may regard the existence of such rules as part of the essential Soar theory, but be non-committal about the precise mechanism by which the rules are applied. This is a perfectly acceptable position if the details of the mechanism are to be regarded as questions for future research, or if they are in principle undecidable.

Natural language descriptions, however, have severe disadvantages. They are potentially ambiguous and of limited precision. This can hinder the derivation of unequivocal predictions. Furthermore, in being abstract, natural language descriptions allow the details of a theory to be filled in in an indefinite number of ways. If such variation does not affect the higher-level properties of the theory, then this may not be of concern, but as Hunt and Luce point out, with reference to the longest, and presumably most detailed, natural language/diagrammatic description of Soar, "Newell's description of Soar is not detailed enough to ensure that any two programmers, having read the book, would produce computationally equivalent programs" [18, p. 448].

2.1.2. Procedural descriptions

Many current psychological theories are highly complex, with the explanations they offer involving the interaction of several components or functional modules. As a result, computational instantiations of theories are often vital in establishing a theory's predictions. Soar is no different in this regard, and the Soar community draws heavily on the behaviour of the Soar implementation when attempting to demonstrate how the theory accounts for empirical results. However, if a program is viewed as a unitary theoretical statement in this way then presumably all of its components and processes must be treated as having equal theoretical status. This seems unsatisfactory. In the case of Soar, for example, there are many features of the program which do not seem to have much theoretical force, such as:

(1) The particular algorithms for matching and firing production rules and removing working memory elements during processing. These are designed primarily for efficient operation on conventional computers. If the Soar architecture is "correct", does the neural implementation also embody these algorithms?

(2) The choice of preference semantics for operator selection, which has been fleshed out in different ways in different implementations [22–24, 31]. It is not clear that any of these formulations is superior, and this is clearly an aspect of the theory which is somewhat fluid. Little seems to hang on the details of different

choices. Indeed it is unclear what sort of evidence could distinguish between different possible semantics. It seems hasty and possibly unnecessary to claim that any *particular* preference semantics is part of the Soar theory, even if some preference semantics is necessary for simulation purposes.

(3) The generalised production language, which allows complex operations (such as input, output and arithmetic) to be triggered directly by productions, rather than, for example, input and output being governed by specialised sensor/effector mechanisms and arithmetic being the product of more primitive processing (as apparently required by the cognitive theory [31]).

These points highlight the most important difficulty with equating a program with a theory: if details are required solely to achieve a complete, executable implementation of the theory then those details may have unpredictable influences on the implementation's behaviour, obscure the essential claims of the theory, and compromise the intentions of the theorist. Given this, any claimed predictions of the theory made on the basis of the implementation must be open to debate, as must the claim that an implementation makes the theory testable. Again, this is a point touched upon by Hunt and Luce: "we want to know whether the simulation was achieved by those parts of the program that embody psychological theory or by parts regarded as choices of convenience in programming" [18, p. 448].

A further concern with procedural descriptions of theories as complex as Soar is that it is difficult for those not directly involved with the theory's design and implementation to contemplate the theory as a whole or to modify (or even isolate) those parts which correspond to specific theoretical assumptions (cf. [10]). In the case of Soar, which now runs to several Megabytes of C code, it is not possible for outsiders who accept most, but not all, of the underlying theoretical assumptions to investigate Soar-like theories by isolating, and systematically modifying, the computational realisation of particular assumptions.

2.1.3. Declarative specifications

A third description method that has been used by the Soar community is the formal specification of the program in the language Z. This is a mathematical language, based on first-order logic, which was designed to provide clear descriptions of what a program should do under what circumstances independent of technology-specific details and to permit formal verification of its correctness [44]. Roughly, a program is decomposed into a hierarchy of operations, each of which is then described in terms of the conditions that must hold before the operation can be executed and the conditions which must hold after execution. A Z specification is not itself an executable program, but rather a clear statement of the required behaviour of the program separate from any specific implementation. In principle, then, it is a description of a theory about which one can reason without worrying about whether it is implemented as a procedural program (e.g., in LISP or C), a piece of hardware, or as nervous tissue.

The purpose of the Z specification of Soar [27] was primarily to provide a sound basis for engineering Soar *qua* computer program rather than Soar *qua* psychological theory. The Z formalisation specifies everything that is necessary for an executable version of Soar. It is not a description at the level of abstraction appropriate for clearly stating the underlying cognitive theory. Consequently, as with the procedural implementations, the Z specification makes no distinction between the essential theoretical elements and processes which appear to be matters of implementation detail (such as those features noted above). If a Z specification of Soar were developed which attempted to capture the psychological theory, it may be possible to make such a distinction, although Z itself does not provide explicit support for it.

2.2. Requirements for a theory specification technique

The methods of theory description surveyed above fail in different ways. Natural language and diagrammatic descriptions allow theories to be stated at the appropriate level of abstraction, but are imprecise. Procedural descriptions are precise, but do not abstract away from implementation details, conflating the theory with its implementation. Formal specification languages may do better, but existing specification languages do not explicitly support the theory/implementation distinction. From these failings we may distill four requirements for a theory specification technique: clarity, succinctness, executability, and theory/implementation separation.

Clarity is required in order to avoid the interpretation/ambiguity problems of natural language and so that theorists may isolate theoretical assumptions. Succinctness also aids the isolation of theoretical assumptions: excessive verbosity is as harmful to clarity as an obscure notation. The case for executability is established by the complexity of current theories, and the difficulties that such complexity present for anyone attempting to derive predictions directly from a specification of a theory. As noted above, it is in general only through simulation that the complex behaviour of a number of interacting assumptions can be determined. Finally, the separation of theory and implementation detail is required if the complex behaviour displayed by an executable theory is to be correctly attributed to theoretical, and not implementational, aspects of the specification. We refer to this separation as the above-the-line/below-the-line (or A|B) division, with theoretically motivated aspects being located above the line and implementation details located below the line.

2.3. Executable specification languages for cognitive modelling

We believe that the failings of current articulation methods can in part be addressed by the development of an appropriate executable specification language. By this we mean a language which may be used to specify a theory in a sense to be elaborated below yet which may also be animated; once specified the model can be executed to check that its behaviour is indeed that which it is claimed to be. Such a language must be precise (a specification language can tolerate no ambiguity) but, where necessary, it should allow one to escape from algorithmic details, specifying what a process does rather than the mechanical details of how it is done. Critically, the language must support the distinction between theory and implementation detail and it is for this reason that the formal specification language Z is inappropriate.

Given the current usage of formal specification languages in computer science, our use of the phrase "executable specification language" requires some elaboration. Van

Harmelen and Balder [45] summarise the purposes of formal specification languages as:

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

While we claim that the work presented below demonstrates that the particular executable specification language we have employed has considerable virtues in respect of the first two points, it is not intended to address the third. The aims of psychologists attempting to construct computational models do not include the mathematical verification of software. We view executable specification languages as *describing* how a cognitive theory will behave, not as *prescribing* how it ought to behave. As argued above, properties which seem of greater relevance to the cognitive scientist are clarity, succinctness, executability, and support for theoretical abstraction.

2.4. A|B decomposition

Executability is essential if the behaviour of complex theories is to be determined, but it has its price. As noted in Section 2.1.2, there are typically many aspects of a cognitive theory whose precise implementation is not relevant to the theory. As such it has been claimed that it is not helpful to implement cognitive theories in this way. In brief, the argument is that executability requires over-specification: the specification of details beyond the scope of the theory. According to this argument, the precise specification of such irrelevant details would only conflate the theory with its implementation and obscure theoretically important aspects. The problem with this argument, of course, is that the core theory itself makes no formal predictions. Typically predictions are made intuitively. However, a careful examination reveals that the point at issue is not one of specifying the irrelevant details of a theory, which executability requires we must do, it is one of keeping implementation details distinct from theoretically motivated aspects. Once implementation details are admitted (but kept distinct from the theory) qualified predictions can be drawn. This is the idea of A|B decomposition: to draw a clear distinction between essential theory and implementation details, thus allowing predictions to be properly drawn.

Now, if implementation details are truly irrelevant, then *any* implementation of those details should be adequate. Thus, a theorist may give an input/output specification of some functional subcomponent without specifying the details of the algorithms or mechanisms which compute the function. From the point of view of the implementation, the subcomponent might just as well be implemented as a look-up table (subject to finiteness considerations). More commonly, the theorist may merely state constraints on outputs given certain inputs, without those constraints necessarily determining a unique output.

Johnson-Laird illustrates this second approach in the context of his theory of syllogistic reasoning [20]. According to this theory, people construct "mental models" of propositions (such as *some beekeepers are not chemists*). These models consist of tokens representing the terms in the proposition, but "the number of tokens [...] is arbitrary" [20, p. 97], and "a crucial point about mental models is that the system for constructing and interpreting them must embody knowledge that the number of entities depicted is irrelevant to any [...] inference that is drawn" [20, p. 98]. So the system which Johnson-Laird proposes for constructing mental models is constrained, but not totally specified. It does not specify how many tokens will be present in any particular model. This style of theorising is valid, provided that the constraints on irrelevant details are articulated with sufficient precision to be testable.

The A|B distinction yields a procedure for testing, via variation below the line, the sensitivity of behaviour to implementation detail, and, therefore, the degree to which aspects claimed to be a matter of implementation are truly irrelevant. Only if behaviour (or those aspects of behaviour under investigation) can be shown to be invariant over a range of alternate implementations of B assumptions, can those assumptions truly be described as implementation details. A|B decomposition effectively delineates a space of related models which share their A component and allows that space to be systematically explored. Such investigations reveal the degree to which high-level assumptions dictate low-level behaviour.

Hinton and Shallice [17] and Plaut and Shallice [38] have explicitly used this approach in the connectionist simulation of the neuropsychological syndrome deep dyslexia. This work involved the characterisation of a space of systems and input/output representations in terms of four general conditions. It was then postulated that the behaviour of any system in the space, when damaged, would have a number of properties, which in fact correspond to those found empirically in the syndrome. A number of specific systems, learning algorithms, and input/output representations which were in different regions of the space were then implemented and it was shown that they generally exhibited the hypothesised properties when damaged.[1] Thus, by varying various details, Hinton, Plaut and Shallice demonstrated that certain aspects of their model could justifiably be classified as implementation details.[2]

It should be clear that the A|B distinction cannot be imposed on an implementation *a priori*. The onus is on the theoretician to declare those aspects of an implementation which are theoretically motivated (A) and those which are implementation details (B). The proportion of A to B in different theories will vary, and theory evolution may well involve the incorporation of aspects previously realised as implementation details into the A component. Nevertheless, the proper application of the A|B distinction, when embedded within a philosophy of science that endorses falsification, requires that aspects of an implementation which are in principle undecidable cannot be regarded as theoretically motivated. In this sense the A|B distinction embodies a hypothesis generating strategy: all aspects of the A component must be empirically testable. Thus, to justify being an A component, an aspect must be critical to the

[1] One of the behaviours initially assumed to be generally predicted as a property of any model defined by the four basic assumptions was actually found to occur only as a result of certain of the below-the-line implementations. It was that the frequency of so-called "mixed errors"—errors which are similar to the stimulus on more than one dimension—is higher than one would expect given the frequencies of errors which resemble the stimulus on a single dimension only. Thus the four basic assumptions corresponded to above-the-line assumptions for only the more global of the behaviours of the model.

[2] Applying the same logic to variations in the preference semantics of Soar (as outlined in Section 2.1.2), suggests that the details of Soar's preference evaluation are also more a matter of implementation than theory.

behaviour of the model as a whole, with variation of the aspect yielding empirically measurable differences in behaviour. The A|B decomposition technique therefore allows us not only to explore supposedly non-significant variant theories within the constraints of a single above-the-line theory, but also to explore related theories within the constraints of related above-the-line specifications. Section 4.3 gives an example of this approach.

We may summarise the above-the-line/below-the-line distinction in the following way. Above-the-line assumptions are theoretically motivated. They should have the following "criticality" properties:

(1) falsifiability: failure of the program to predict the observed behaviour should indicate the falsity of one or more of the A assumptions; and
(2) uniqueness: distinct A assumptions should predict distinct behaviours.

In contrast, below-the-line details are not theoretically motivated. Their effect on behaviour should be subject to the following "sensitivity" properties:

(1) invariance: changes to the B details should have no effect on the behaviour of the program; and
(2) non-compensatability: it should not be possible to compensate for failures of the A assumptions by adjustment of the B details.

In other words, behaviour should be robust with respect to variation of B details but brittle with respect to variation of A assumptions.

The properties of criticality and sensitivity are idealisations: in reality they cannot be categorical. In practice it may be more appropriate to develop indices of criticality and sensitivity so as to quantify the contribution of A and B components to the behaviour of a program. Such an approach has at least two advantages. Firstly, it would allow the sufficiency of competing theories to be compared—a theory with a higher criticality/sensitivity ratio may be regarded as superior to one with a lower ratio given that the theories address the same data. Secondly, it would allow the progress of a research programme to be quantified, with theoretical advances being measured in terms of increases in the criticality/sensitivity ratio.

3. Sceptic: an example executable specification language

The particular executable specification language which we have been working with is Sceptic [9, 16]. Sceptic was designed to combine the capabilities of an executable specification language with capabilities that would facilitate the practical implementation of various simulation and AI applications. Though it was not explicitly designed as a cognitive modelling language, its design was loosely influenced by earlier modelling work using production rule systems. It was considered as a first approximation to the required modelling tool for several reasons:

(a) It had been found to have properties appropriate to the implementation of complex, time-evolving systems (including biological process simulations [48] and logical reasoning mechanisms [16]), autonomous systems [14], planning systems [36] and control processes in image interpretation and medical problem solving [46].

(b) The language interpreter supports a number of mechanisms that are commonly assumed in cognitive theories, including pattern-directed processing, content addressable memory retrieval/update, sequential processing and parallel data propagation modes.
(c) It has a simple but expressive syntax which assists clear and succinct representation of data structures and processes, and its execution model is simple and easily described.

3.1. The Sceptic language

Sceptic was originally formulated as an extension to Prolog (e.g., [6]), in which it is implemented. The primary additional device is a forward chaining control structure (the conditional rewrite rule). Crucially from the perspective of cognitive modelling, this control structure allows procedural control aspects of a program to be expressed in terms of rewrite rules and distinguished from purely declarative aspects expressed in standard Prolog.

Central to any Sceptic program is a database—the system's state—which is queried and modified as execution progresses. Rewrite rules test the state and, on the basis of such tests, either modify the state or trigger further rewrite rules. A Sceptic program consists of a specification of an initial state and a set of rules which specify how the state evolves over time, or how the state changes in response to particular events.

3.1.1. Syntax

Sceptic inherits much of its syntax from Prolog. All data is represented via Prolog terms, which consist of a relation and zero or more parameters, e.g.:

```
block1    augmentation(block1, colour=red)    production(p&q, r)
```

The last example above, `production(p&q, r)`, might represent a production rule, with the arguments p&q representing antecedent matches against working memory (p and q) and `r` representing the consequent creation of a working memory element.

Conditional rewrite rules have the following syntax:

```
InTrigger:
    Condition1, ..., ConditionM
 => OutTrigger1, ..., OutTriggerN.
```

`InTrigger`, `Conditions` and `OutTriggers` are all Prolog terms, possibly containing parameters in the form of Prolog variables (indicated as capitalised strings: `augmentation(Object, Attribute=Value)`, for example, contains three variables).

Two types of condition may be distinguished. Firstly, there are those which test whether or not some condition holds in the current program state. We refer to these as state testers. Secondly, there are generators—conditions which instantiate variables by querying the state, thus generating information. We also distinguish two types of trigger: atomic triggers, which are non-decomposable, built-in, operations such as output primitives and database modification primitives; and compound triggers, which are defined in terms of conditional rewrite rules.

3.1.2. The execution model

The execution of a Sceptic program is governed by a stack which holds all unprocessed triggers. Execution is initiated when a trigger pattern is issued at the Sceptic prompt. This pattern is pushed onto the stack, which is then processed item by item. As each item is taken off the stack it generates a trigger event. If a trigger event is a primitive operation (e.g., a primitive database modification) then it is executed directly. Otherwise it is expanded. If all the conditions of a Sceptic rule (on the left-hand side of the rewrite symbol, `=>`) are satisfied at the moment a `Trigger` event occurs then all the elements on the right-hand side are generated as the expansion of the trigger and pushed onto the stack. Execution of a Sceptic program therefore consists of an indefinite number of cycles (trigger event, check conditions, process actions) which continues until the stack is empty.

Sceptic variables are bound by assigning values during the triggering step or when checking left-hand side conditions. Such conditions may either match an explicitly stored term, in which case the variables simply take the values of the constants in the same position of the matching term, or generate values from the program state. Right-hand side variables take the values of any variables to the left of the rewrite symbol which have the same name. Variables in the conditions of rewrite rules are universally quantified so that if Sceptic can find multiple solutions for such conditions, multiple copies of actions will be pushed onto the stack with appropriate variable bindings.

To illustrate, in the following Sceptic rule:

```
time(Time):
    input_buffer(Object, Att=Value)
  => add_wm(Time,Object, Att=Value).
```

a timestamped data item is added to working memory whenever a `time` event occurs (e.g., a clock ticks) and there are data in an input channel. If n items are in the input buffer when the `time` trigger occurs then n `add_wm` items will be created and pushed onto Sceptic's stack.

As noted above, Sceptic was originally developed as an extension to Prolog, and employs built-in Prolog operations to check and update the program state, carry out input/output operations, etc. However, this is not considered to be fundamental. These operations could be implemented with any general programming language, by specialised hardware, or even perhaps by a distributed, subsymbolic mechanism. The essential aspects of Sceptic are the conditional rewrite rule and the division this engenders between conditions and actions.

3.2. A simple example

Imagine we are constructing a cognitive theory, call this COG1. COG1 is a rather simple theory, though perhaps it captures a motif common to many real psychological theories of twenty or so years ago. COG1 can be described in English as follows:

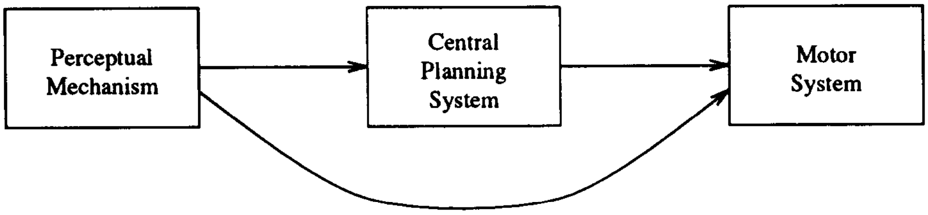

Fig. 1. A box and arrow diagram of COG1.

Information enters a central cognitive system through a perceptual mechanism, from where it is passed to a central processing channel in which actions are planned and executed. Since certain kinds of sensory information must be processed urgently, however, there is a "fast" link, between the perceptual mechanism and the action execution mechanism, which operates in parallel with the limited capacity central channel.

COG1 might have been presented by a theorist in a box and arrow notation, such as that in Fig. 1. In the box and arrow notation boxes are typically used to represent hypothesised functional modules, and links between the boxes represent the flow of data or control information between modules. Of course the box and arrow notation does not represent the whole of the intended theory, just an abstraction in which the detailed mechanisms of the modules and communication channels are disregarded. They are not considered to be part of the theory at this level of description.

COG1 might be specified in Sceptic by the following three conditional rewrite rules:

```
perceptual_event(Event):
    direct_action(Event, Action)
 => execute_motor_command(Action).
perceptual_event(Event):
    not(direct_action(Event, Action))
 => construct_and_execute_plan(Event).

construct_and_execute_plan(Event):
    central_planning(Event, Action)
 => execute_motor_command(Action).
```

The example illustrates two points about the structure of a Sceptic specification. Firstly, each rewrite rule corresponds to one arrow in the box and arrow specification. The first rewrite rule, for example, corresponds to the arrow from the perceptual mechanism to the central planning system. Secondly, the three triggers employed (`perceptual_event`, `construct_and_execute_plan` and `execute_motor_plan`) correspond to the three boxes. With regard to processing, when a triggering event occurs (e.g., when, in COG1, there is a perceptual event), the conditions of all corresponding rules are evaluated in parallel against the state of the system. If the conditions evaluate to true, then the consequent triggering events are generated in series, resulting in a further burst of processing. Consequent triggers may also modify the system's state.

Not shown above are the details of the antecedent conditions. These may be specified in standard Prolog. In particular, if the domain is restricted, each valid instance may simply be listed. Note that such conditions are not detailed in the box and arrow diagram, though they are necessary if the theory is to be animated.

This example also shows some of the expressive power of rewrite rules. Although it may not be immediately obvious, conditional rewrite rules can directly encode all standard control structures. The rules for `perceptual_event` demonstrate how a standard "if ... then ... else ..." structure may be represented. All standard loop control structures may be implemented by recursive rewrite rules: rewrite rules whose trigger appears on their right-hand sides, i.e., rewrite rules which trigger themselves. Furthermore, because variables in conditions are implicitly universally quantified, and because all instantiations of the conditions of any trigger are evaluated with a constant state, the rules embody a form of parallelism: `central_planning` may yield multiple actions for any given event. These will be executed sequentially in the order they are pushed onto the stack, but the evaluation of what actions to perform is effectively performed in parallel.

3.3. A|B decomposition in Sceptic

It was noted above that the structure of the Sceptic specification of COG1 mirrored the box and arrow diagram, although the details internal to the boxes were not supplied. Given that such details were not specified in the original statement of the theory, the Sceptic specification accurately captures the theory at the box and arrow level, i.e., at the level of psychological theorising. Details of the two generating conditions, `perceptual_event` and `central_planning` are necessary if the specification is to be executed, but they are not part of the psychological theory. In this way, Sceptic is able to support the theory/implementation division via syntactically and semantically distinct structures within the language: above-the-line mechanisms may naturally be expressed in terms of conditional rewrite rules, with below-the-line detail being expressed in terms of Prolog predicate definitions.

The use of Prolog predicate definitions to specify the conditions under which Sceptic rewrite rules are triggered is not, as noted above, an essential part of Sceptic. Nevertheless, Prolog, when interpreted as a logic programming language in the strongest sense, is particularly apt here, as it allows a purely declarative statement of implementation detail: below-the-line aspects may be stated in terms of constraints that must hold, rather than as a set of algorithms. This avoids any possibility of attributing algorithmic properties to implementation detail, mirroring the fact that the implementation in algorithmic terms of below-the-line aspects is, by definition, beyond the bounds of theory.

4. An assessment of Sceptic

Our goal is to develop a specification language in which definitions of models are readable and succinct, modular, appropriately abstract, and yet still executable. In order to evaluate Sceptic in these respects we now present (a) a complete reconstruction of Soar 4, demonstrating the expressive power of Sceptic; (b) an overview of Sceptic specifications of a number of other theories, demonstrating the generality of Sceptic; and (c) details of several computational experiments performed with the Sceptic specification of Soar 4, demonstrating empirical methodology based on the A|B distinction used with Sceptic.

4.1. The Sceptic reconstruction of Soar

The toy psychological model above illustrates the basic concepts of Sceptic, but to evaluate the language on a more challenging example we have reconstructed a version of the Soar architecture. Soar was chosen for this purpose for several reasons. Firstly, in many respects Soar is, at the present time, the most established and mature computational theory within the symbolic paradigm. Secondly, Soar represents a major challenge to the development of a cognitive modelling methodology because of the range of phenomena it addresses and the technical complexity of the program. Soar is the product of more than a third of a century of research, beginning with a narrow focus in problem solving [33], and progressing to an "architecture for general intelligence" [23], and even a "unified theory of cognition" [31]. Finally, the Soar program has been distributed to many research groups who are investigating and contributing to the Soar theory; a deeper understanding of the theory is therefore more widely disseminated than for most computational models. Version 4 of Soar was chosen because this is the version addressed by most existing verbal descriptions (e.g., [23,31]). In addition it is sufficiently complex to illustrate the use of Sceptic, and, as argued by Cooper and Shallice [10], it is not clear that the additional complexities present in later versions have any deep psychological motivation.

Irrespective of Sceptic's precise status as a *general* executable specification language, it is suitable as an implementation language for Soar and similar cognitive theories for several independent reasons. Firstly, the language incorporates Prolog's pattern matching capabilities. Analogous capabilities are implicated in Soar and many other current theories of cognitive processes. A second feature of Sceptic is its functional parallelism. The division between conditions and actions allows all conditions relating to the expansion of any trigger to be effectively evaluated in parallel. This is particularly useful in specifying Soar's elaboration phase, which involves many functionally parallel activities. Thirdly, and perhaps most appropriately for Soar, Sceptic's conditional rewrite rules bear a close resemblance to the productions which form the basis of Soar's elaboration cycle.

Within the Sceptic specification of Soar each of the main components (elaboration, decision, subgoaling and chunking) is specified in terms of a set of Sceptic rules. The complete set of these rules is contained in Appendix A. The trigger of each rule is used to represent the communication between the components. The conditions of the rules test the state of the Soar system, notable working memory, preference memory, and so forth. When certain states are found to hold the actions of the rules are executed. These actions implement both direct alterations to the state of the system, such as additions to working memory, and the sending of messages to other system modules by means of triggers. Where the literature on Soar warrants it (i.e., it is clear that the internal steps of the component's operation are part of the psychological theory embodied in Soar) the internal structure of the component is implemented above the line by introducing a collection of Sceptic rules which specify the internal details of the component. When there appears to be no theoretical justification for elaborating the operations in detail no such decomposition is attempted—they are left below the line.

Our aim in focusing on Soar was not simply to reimplement it, but to rationally reconstruct it. This was successful in that by using trace functions which duplicate the surface form of the original Soar's output we have been able to demonstrate that, on published examples, the behaviour of the Sceptic version is identical to the original. However, because of the emphasis on rational reconstruction, there are certain aspects of the Sceptic version which are not merely reimplementations of the corresponding aspects of the LISP implementations. Firstly, the decision phase is decomposed into two separate processes, one of deciding upon a certain context element and one of bookkeeping. Secondly, the reconstruction highlights architectural differences behind the classification of impasse types. In our view these differences actually increase the clarity of the theory's structure whilst preserving its behaviour.

Of some interest from the methodological point of view is that comparison of the LISP and Sceptic versions of Soar 4 shows that Sceptic Soar (including both A and B components) is less than 10% of the size of the LISP version measured in lines of code. This difference cannot be entirely attributed to the expressive capabilities of Sceptic—the Sceptic version does not include theoretically irrelevant algorithms (such as the RETE algorithm for efficiently firing productions), nor does it include many of the interface facilities designed to make the LISP implementations more user-friendly. We have not attempted to duplicate such features, since our goal is only to include those theoretically significant features of the program together with just those below-the-line mechanisms which have to be implemented to make the program executable. Nevertheless, the reduction in size does demonstrate that all that is necessary for an executable statement of the theory can be given in substantially reduced terms.

Although the complete implementation is substantially reduced in size, at 2000 lines of code it is still a substantial program. It is hardly a succinct representation of the Soar theory, and this has obvious implications for difficulty of understanding by those other than the authors. However, the A|B distinction serves us well here. The A code consists of only 28 Sceptic rules. If we accept the A|B decomposition and equate the essential Soar 4 theory (see [32, p. 467]) with the above-the-line rewrite rules, then this represents a dramatic reduction in complexity. Notwithstanding the minor differences between the two programs noted above, we consider this reduction to be important. The relative ease of understanding of triggered condition-action rules, as compared with arbitrary programs, also contributes to clarity. The Sceptic program, and hence the underlying model, is in our view much easier to understand, criticise, assess and modify than a program written in a conventional programming language.

Furthermore, paralleling the Sceptic specification of COG1, there is also significant modularity. Individual Sceptic rules capture the functionality of individual components of the Soar architecture. The rules can be modified to yield different behaviour without requiring changes to other parts of the program, though of course such changes cannot be arbitrary and must be made with care and attention to the data and control structure of the global program. The extent of the modularity of the Soar specification, and some of the consequent benefits, are illustrated by the computational experiments discussed in Section 4.3.

4.2. The generality of Sceptic

Even if our optimistic conclusions based on our reconstruction of Soar are justified, one may reasonably ask to what extent Sceptic embodies a general method, and to what extent the benefits may reflect merely a good match between the needs of the Soar architecture and the capabilities of Sceptic. Other work suggests that this is not the case. A variety of cognitive theories have been implemented using Sceptic, including Johnson-Laird's theory of model-based reasoning [20], Morton et al.'s "Headed Records" theory of memory structure and recall [28], Sloman's model of motives and emotions [43], Barnard's theory of Interacting Cognitive Subsystems [4] and Norman and Shallice's model of automatic action control [35].

Johnson-Laird's model-based theory of human syllogistic reasoning [20] is essentially an algorithmic one, specifying a number of processes claimed to be involved in syllogistic reasoning. These processes include building a model which satisfies a pair of premises, inspecting a model for a putative conclusion, attempting to refute a conclusion by manipulating a model, etc. The theory has previously been implemented in LISP, and, at an abstract level, is well specified in the literature. The aim of the Sceptic specification [7] was primarily to provide an executable version of the theory which separated theoretical statements and implementation details. This was achieved by associating one Sceptic trigger with each process described by Johnson-Laird, and directly translating the LISP implementation of the remainder of the model into Prolog generators and state testers. The resulting implementation replicated the behaviour of the LISP implementation on all 64 configurations of syllogism premises.

A further processing theory for which a Sceptic specification has been developed is Morton et al.'s Headed Records theory of memory structure and recall [28]. This theory claims that memory is structured in terms of a set of discrete records, with each record consisting of a heading consisting of a set of access cues and a body containing the content of the memory. Recall of any particular record may involve a sequence of recall cycles where an inappropriate record is accessed and its content is used, together with external cues, to construct a new key for a further recall cycle. This theory, though not previously implemented, was well described in the literature and proved to be directly implementable in Sceptic. The A component of the complete recall cycle was coded with just five triggers, corresponding to the five principal processes: create cue set; search record heading; retrieve matching record; evaluate retrieved record; and extend cue set. Eight conditions were necessary to encode the B component.

A very different style of theory is represented by Sloman's theory of motive processing [43]. This theory concerns the scheduling and attempted satisfaction of multiple, possibly conflicting, goals, and consists of a number of parallel communicating processes. Such interacting processes can be modelled in Sceptic in terms of transition rules for a single global database, where the database elements represent the data on which the interacting processes operate [8]. This data-flow style of specification is very different from the control-flow style adopted for Soar, syllogistic reasoning, and memory recall, where the flow of control is specified in terms of Sceptic rewrite rules. In particular, the data-flow style is purely declarative, more closely approximating formal specification. The specification of Sloman's motive processing architecture includes just three rewrite rules: one to add new database elements; one to transform current database elements; and one to delete database elements. Each rule has numerous instantiations. Thus, for example, one instantiation of the transform rule states the conditions under which a suspended motive will become active, while another instantiation specifies the reverse transformation. The three rules can be viewed as an engine for executing the actual specification which is given in fully declarative Prolog. This specification also scores well in terms of clarity, succinctness and modularity. In addition, the A|B distinction is manifest in the separation between instantiations of rewrite rules and the Prolog conditions on which they draw.

A second data-flow theory which has been explored is Barnard's cognitive architecture based on Interacting Cognitive Subsystems [4]. Again this is a theory which models cognition in terms of several interacting processes, and again the Sceptic specification is driven by just three rewrite rules. In fact, the same three rewrite rules were used to harness both Sloman's motive processing theory and Barnard's Interacting Cognitive Subsystems. Full details of all instantiations of the rules for Barnard's system have not been developed, but exploratory work suggests that the specification style applied to Sloman's motive processing model generalises to a class of theories based on concurrent asynchronous communicating processes.

Lastly, Sceptic has been applied to Norman and Shallice's Contention Scheduling theory of action selection [35]. This concerns the sequencing of actions and the selection of objects involved in those actions within segments of automatic or well-learnt behaviour, such as preparing breakfast or driving a car along a familiar route. The heart of the theory is an interacting activation network of schemata or action segments. The theory differs substantially from those above in that it includes continuous valued variables representing the activation levels of schemata. Nevertheless, Sceptic has again proved capable of accommodating the theory both clearly and succinctly [11]. The style of specification follows that used for the theories of Sloman and Barnard, specifying database elements and rules for generating/updating/deleting those elements. The continuous valued variables are treated as one further component of the database elements, with the updated values being calculated from the current values according to standard interacting activation difference equations.

The size of the Sceptic implementation of Soar, together with that of each of the above theories, is shown in Table 1. Whilst the figures can only be taken as giving an approximate guide to the complexity of the implementations, it is clear that all implementations are concise, though Motive Processing, Contention Scheduling and Interacting Cognitive Subsystems require further domain-specific code to be executed.

One obvious class of theories for which the approach does not seem directly useful is distributed or connectionist theories. A highly symbolic approach is obviously inappropriate for implementing a subsymbolic theory but Sceptic may still have utility in

Table 1
The size of various Sceptic implementations

	A: rewrite rules	B: conditions
Soar (Version 4)	28	109
Mental Models	8	50
Headed Record Recall	5	8
Motive Processing	20	15
Interacting Cognitive Subsystems	3	4
Contention Scheduling	11	46

exploring hybrid models by implementing symbolic components as Sceptic rules, and the subsymbolic components as below-the-line procedures. This has been done in an exploratory study of medical diagnosis [13]. A back-propagation network was trained using a collection of records of patients with different diseases presenting with different sets of symptoms. The net learned a discriminating set of weights in the usual way. Another below-the-line procedure monitored the behaviour of the net and generated symbolic rules which were stored by a Sceptic program. As new cases led to weight revisions new or different rules were generated. The Sceptic component maintained a consistent set of rules using a simple truth maintenance mechanism [16].

Our conclusion is that the implemented theories encompass a wide range of theoretical traditions and the relative ease of implementation suggests that the approach has considerable generality.

4.3. Sceptic as an experimental tool

In Section 2.4 we argued for the utility of sensitivity and criticality analyses in evaluating the theoretical and implementational aspects of a model. The clarity and succinctness of the Sceptic specification of Soar opens up the possibility of conducting such analyses on Soar. In particular, with regard to criticality, we may consider the effect of modifying specific theoretical assumptions on the behaviour of the theory as a whole. In order to demonstrate the methodology, and to prove the utility of computational experiments, we consider here three experiments with the working memory component of Soar 4. Each experiment was conducted by modifying the Sceptic implementation of Soar 4 and then performing a number of simulations to evaluate the effects of the modifications. In order to emphasise that computational experiments are in principle no different to other experiments, each is reported in a style similar to that used to report more standard experiments.

4.3.1. Experiment 1: the single state principle

Rationale: In 1990, the Soar community moved from version 4 to version 5 of their architecture. The transition to Soar 5 was motivated primarily by a psychologically unrealistic requirement that Soar 4 placed on working memory, namely that the sequence of states traversed within a problem space is maintained in working memory until the relevant goal is achieved. This aspect of Soar 4 meant that as a problem space was traversed, the working memory requirement increased at a virtually constant rate. To counter this, Newell [31] introduced the single state principle (SSP): within any problem space, only a single state is maintained in working memory at any time. In Soar 5 this is effected via destructive state modification: as each operator is applied the current state is destructively modified to yield a new state. This destructive state modification, however, results in enormous complications within Soar. Experiment 1 was designed to demonstrate that a version of the SSP could be incorporated into Soar 4 without those complications. The idea was that old context elements (i.e., problem spaces, states, and operators) could simply be deleted from working memory when they were superseded. This is not destructive state modification—the old state is deleted rather than being destructively modified—but it is a version of the SSP.

Method: The Sceptic specification of Soar 4 (as given in Appendix A) was augmented with further rewrite rules which effected the deletion of superseded working memory elements whenever a new context element was installed. This situation is complicated by the fact that working memory is a form of truth maintenance system, with elements being justified by the instantiations of the productions which led to their creation. Therefore, when a working memory element was removed, its justification was also removed from instantiation memory. The Sceptic rules are as follows:

```
install_context_object(Goal, Attribute, NewValue):
    parameter(single_state_principle, true),
    wm_match(augmentation(goal, Goal, Attribute=CurrentValue))
 => remove_superseded_wmes(CurrentValue, Goal, Attribute).

remove_superseded_wmes(Identifier, Goal, Class):
    wme_to_be_removed_after_installation(Identifier, Goal,
                                         Class, WME)
 => wm_remove(WME).
remove_superseded_wmes(Identifier, Goal, Class):
    wme_to_be_removed_after_installation(Identifier, Goal,
                                         Class, WME)
    generating_instantiation(WME, Instantiation, FiringGoal),
    instantiation_not_supported(Instantiation)
 => im_remove(Instantiation, FiringGoal).
```

The first of these rules is triggered whenever a context object is replaced. It finds the current context element by matching against working memory and removes any elements connected to this but not connected elsewhere to the context stack. For each such working memory element, if the production instantiation responsible for the element's presence in working memory is no longer applicable, it too is removed.

Simulations were run, both with and without the SSP, on two tasks: the standard monkeys and bananas task, as released by CMU with Soar 4.5, and a modified version of this task in which the monkey only considers operators whose preconditions are satisfied. A trace of decisions made, productions fired, and working memory size, was kept for both conditions for each task.

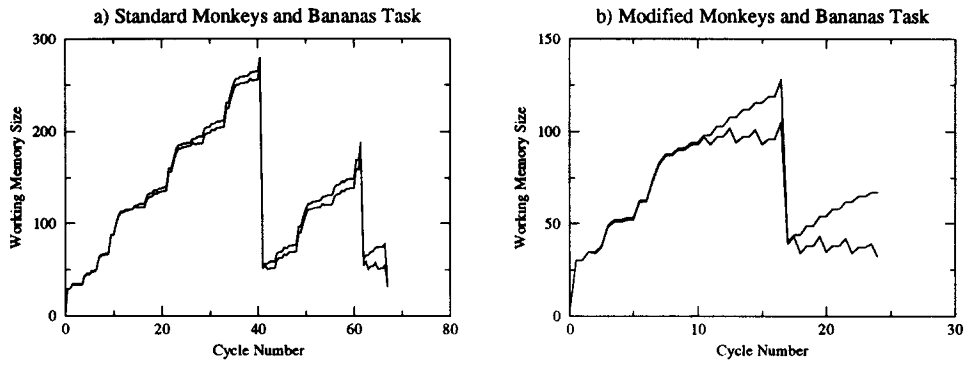

Fig. 2. The effect of the single state principle in Soar 4 on the size of working memory.

Results: The graphs in Fig. 2 show the size of working memory versus decision number for each of the tasks. In each case the higher curve corresponds to the original Soar 4. The lower curves were obtained when the above version of the SSP was incorporated into the Sceptic specification. A comparison of the decisions made and productions fired revealed that there was no difference in behaviour between the two versions of Soar 4 on each of the tasks.

Discussion: There is little difference between the working memory requirements of the two versions of Soar on the standard monkeys and bananas task. This is because in this task the problem solving behaviour consists primarily of a succession of impasses. Each impasse results in a subgoal which requires approximately 30 more working memory elements, irrespective of any version of the SSP. The difference in working memory size is shown most clearly when problem solving is progressing within a single problem space, when many decisions involve the selection of a new state after the application of an operator. At these times, the working memory requirement of the standard theory increases approximately linearly with the number of decisions, whereas the working memory requirement of the modified theory remains approximately constant. This can be seen most clearly in cycles 10 to 16 and 17 to 24 of the modified monkeys and bananas task. The results of the experiment thus question the original rationale for Soar 5 since it appears that a viable alternative to destructive state modification exists which overcomes the problems of increasing working memory size and yet which is substantially simpler than that employed in Soar 5.

That there was no difference in the sequence of decisions made by Soar 4 with and without the SSP would seem to indicate that the SSP is not a critical (i.e., above-the-line) assumption. It is only if behaviour is widened to include working memory size that crit'cality can be seen. Given this, it is necessary to show that the working memory requirements of Soar 4 with the SSP and Soar 5 are equivalent if we are to be sure that the additional complications in implementing the SSP in Soar 5 are implementation details. We have been unable to establish this in the absence of an executable specification of Soar 5.

4.3.2. Experiment 2: working memory decay

Rationale: A common assumption within cognitive psychology is that working memory is subject to decay or interference, in which there is spontaneous loss of information over time (see, e.g., [3]). Even so, human problem solving is relatively robust. Can Soar tolerate reasonable interference or loss from working memory? Experiment 2 was designed to explore Soar's behaviour under conditions of working memory decay.

Method: The elements of Soar's working memory are arranged in the form of a connected, directed, graph. A pilot experiment revealed that, due to the possibility of this graph becoming disconnected, unconstrained working memory decay frequently lead to serious and irrecoverable problems. Thus, in this experiment decay was restricted to terminal working memory elements. This restriction on decay might be justified in terms of the peripheral nature of these elements: they are less well integrated into working memory, and hence, more liable to decay.

The Sceptic specification of Soar 4 was thus augmented with a further rewrite rule which effected a random decay of working memory elements. A parameter specified the degree of decay. The rule, triggered at the end of each decision cycle, is as follows:

```
memory_decay:
    parameter(working_memory_decay, D),
    wm_match(WME),
    wme_is_terminal(WME),
    random(Z),
    Z < D
  => wm_remove(WME).
```

This rule specifies that a working memory element should be deleted if it is a terminal element and if a pseudo-random number between 0 and 1 generated for that working memory element is less than the decay parameter.

The simulation was run 20 times with different random seeds on the two tasks used in experiment 1 for values of the decay parameter ranging from 0.000 to 0.010 at intervals of 0.001. A decay of 0.001 amounts to a working memory half life of 693 cycles or 42 seconds at 60 msec per cycle; cf. [31]. A decay of 0.010 amounts to a working memory half life of 69 cycles or 4.1 seconds at 60 msec per cycle. Execution was terminated if the goal was not achieved within 100 decisions. A trace of decisions made, productions fired, and working memory elements decayed, was kept for each run.

Results: On analysing the traces of each run, two broad sorts of behaviour could be distinguished. Either Soar would successfully complete the task, with seemingly perfect problem solving (i.e., choosing exactly the same problem spaces, states and operators as when running without working memory decay), or Soar would subgoal apparently aimlessly. In the second case, some critical working memory element would often eventually decay, resulting in complete breakdown (see below). Fig. 3 shows the number of instances of each sort of behaviour.

The results show two trends. Firstly, as the decay rate increases, the rate of success drops off gradually: Soar does not always simply collapse when working memory decay

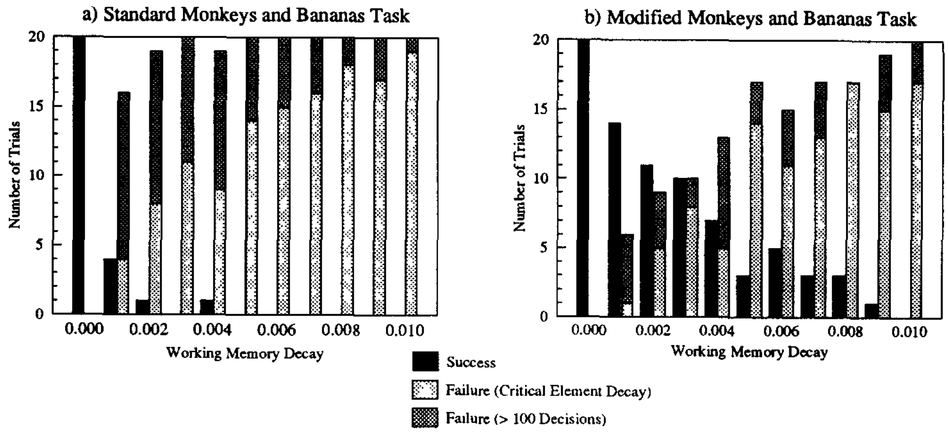

Fig. 3. Behaviour of Soar with decay of working memory leaf elements.

is introduced. Soar may succeed in short tasks even with some working memory decay. This is most clear in the case of the modified monkeys and bananas task, which involves only 24 decisions. This task was correctly solved on one trial out of twenty when the decay rate was 0.009 per cycle, corresponding to a half life of 4.6 seconds. The effect of decay is more pronounced when Soar is attempting longer tasks. The standard monkeys and bananas task requires 67 successive correct decisions. Here, substantially less decay can be tolerated.[3]

The second trend concerns the number of decisions made before a critical working memory element decays, and is revealed most clearly by the standard monkeys and bananas task. Though not shown by the graphs, experiment 2 confirms that, as the working memory decay rate increases, the number of decisions before such an element decays decreases.

Discussion: It has been shown that Soar can withstand some working memory decay, and in this sense it is not entirely brittle. However, with an intermediate decay rate (say 0.002, giving a working memory half life of 21 seconds) Soar is still generally unable to string together 67 decisions, or 4 seconds of behaviour as required by the standard monkeys and bananas task. Soar's behaviour thus appears far more brittle than that of the average human. However, the productions employed in these tasks were designed to be optimal. They perform no error checking and there is no redundancy in their actions. Given that an alternate way of looking at this experiment is as a criticality analysis of working memory elements—if very little decay can be tolerated then most working memory elements must be critical to the ideal problem solving behaviour—

[3] This reasoning, and the fact that according to the cognitive theory Soar is continuously making decisions and does not halt when it completes some task, suggests that the results might be more appropriately presented in terms of the mean length of sequences of correct decisions. We have not attempted this here as these experiments are intended only as a demonstration of our methodological prescriptions, and the difficulty of extracting such information by hand would not be justified.

we may interpret the results as measuring production redundancy.[4] However, building redundancy or error detection and recovery into the productions would further increase the capacity required of working memory.

These results raise a general question concerning possible dependencies on implementation details. Could details which we have taken to be below the line influence Soar's behaviour when above-the-line assumptions are modified? Unfortunately this appears to be the case. Built into the implementation details of the decision cycle are various assumptions concerning the structure of working memory. One of these is working memory connectedness. This assumption is satisfied by the restriction of decay to terminal elements. A second assumption, which may be violated in this experiment, is that each goal has a problem space, state, and operator. Because of these implementation dependencies, decay of these elements leads to complete breakdown: the decision procedure fails if a problem space, state, or operator decays. Presumably it would be more desirable for Soar to attempt to redecide any missing elements. This demonstrates the importance of being explicit about assumptions built into implementation details. Such assumptions amount to constraints on details below the line, without forcing a unique below-the-line specification.

4.3.3. Experiment 3: instantiation memory decay

Rationale: It could be argued that the previous experiment did not correctly take account of Soar's production system base: if a working memory element decays, then the production which originally put the element into working memory can simply fire again, replacing the decayed element. Experiment 3 was designed to test if Soar is more robust under these conditions, by allowing instantiation memory (the memory which records production firings) to decay.

Method: The Sceptic specification of Soar 4 was augmented with a rewrite rule which effected a random decay of instantiation memory elements together with the working memory elements which they created. A parameter specified the rate of decay. The rule specifies that an instantiation memory element should be deleted if a pseudo-random number between 0 and 1 generated for that instantiation memory element is less than the decay rate. If an instantiation memory element is deleted, all working memory elements created by that instantiation are also deleted. The conditional rewrite rule is as follows:

```
memory_decay:
    parameter(instantiation_memory_decay, D),
    im_match(Instantiation, FiringGoal),
    random(Z),
    Z < D
 => im_remove(Instantiation, FiringGoal),
    remove_dependents(Instantiation).
```

This rewrite rule was triggered at the end of each decision cycle.

[4] We thank Richard Young for pointing this out to us.

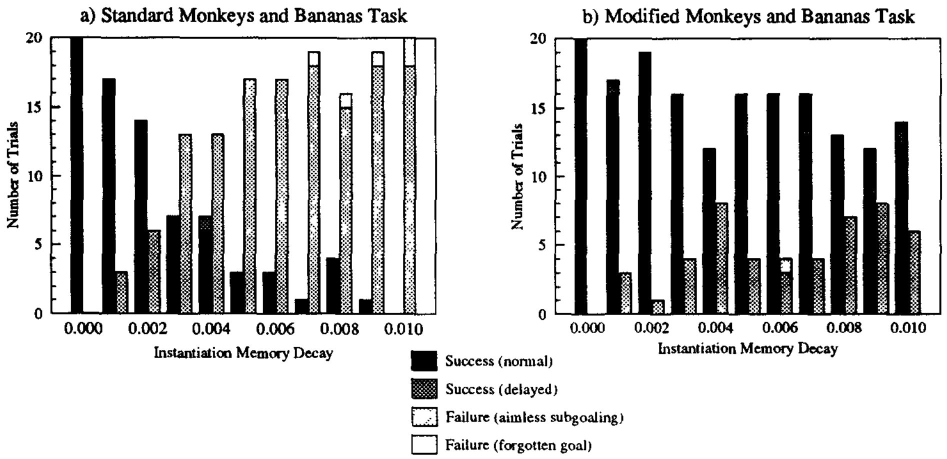

Fig. 4. Behaviour of Soar with instantiation memory decay.

The simulation was run 20 times (with different random seeds) on the two tasks used in experiment 2 for each value of the decay parameter, which again varied from 0.000 to 0.010 by intervals of 0.001. Execution was terminated if the goal was not achieved within 100 decisions. A trace of decisions made, productions fired, and working memory elements decayed, was kept for each run.

Results: The results are summarised in Fig. 4. Once again, performance with memory decay falls into a number of broad classes, though interestingly the classes resulting from instantiation memory decay differ from those resulting from working memory decay. Again, there is the possibility of perfect behaviour, even with substantial instantiation memory decay (0.010 per cycle). As might be expected, perfect behaviour became less frequent as the decay parameter was increased. However, unlike working memory decay, instantiation memory decay can lead to delayed success, where Soar completes its task but requires more decisions to do so. In effect Soar can become sidetracked, or forget some information, but later recover. This sort of behaviour is most frequently shown with substantial degrees of decay in the modified monkeys and bananas task. In general, the delay is relatively small requiring perhaps 5 decisions more to complete the task which normally requires 24 decisions (i.e., of the order of 20%), but one case with decay at 0.010 showed a delay of 170%, eventually correctly solving the 24 decision task in 65 decisions. Closer examination of this case revealed that Soar repeatedly forgot the results of a subgoal before those results could be used, and had to repeatedly recalculate the results.

Discussion: Where Soar failed to complete the task, two sorts of breakdown could be distinguished. Frequently Soar appeared to subgoal aimlessly. In other cases, Soar forgot its goal, and settled into a "waiting mode", requiring new input to trigger further problem solving. In the case of aimless subgoaling, *post hoc* probabilistic analysis suggests that, given time, Soar would always either revert to this waiting mode or succeed.

The differences between the behaviour of Soar 4 with and without instantiation memory decay shows that this form of decay, like the working memory decay of experiment 2, is a critical assumption. As a general memory decay option, however, instantiation memory decay appears to be more psychologically plausible in so far as the resulting system is more robust and one type of performance degradation—delayed success as a result of having to repeat steps—is intuitively plausible. Indeed, this form of behaviour, which was not expected, appears to accord with perseveration (i.e., repeated solving of the same goal) shown in Newell and Simon's cryptarithmetic protocols [34]. It therefore suggests that instantiation memory decay within production system models may be a mechanism that warrants further investigation as a possible explanation for perseverative behaviour in problem solving. This is a novel result generated from the computational experiment: it has not, to our knowledge, been suggested before as a possible explanation of such behaviour.

5. Executable specification languages and modelling methodology

The work described here represents a step towards the rigorous modelling methodology that we would like to see developed in cognitive science, but any modelling language can only form one component of a general methodology. One possible role for such a language is as the computational engine within a software development environment designed to support all aspects of theory evolution. The environment should support computational experiments carried out by means of a modelling language such as Sceptic, though we anticipate that more sophisticated executable specification languages will emerge in the future. The language should permit variations on a model to be investigated and allow criticality/sensitivity analyses, and the environment should provide tools for designing and documenting variations to the model, and for collecting and analysing data generated by such experiments.

Preliminary analysis suggests that such an environment might incorporate a Lakatosian philosophy, of the sort that Newell advocates [31], with a clear division between the hard core of a theory and its auxiliary hypotheses. Within such an environment theory development would be concerned with adjusting the auxiliary hypotheses corresponding to the below-the-line aspects of the model and incorporating assumptions from there into the hard core which corresponds to the above-the-line aspects of the model.

Such a proposed framework for theory development may be too constraining. Although in his main exposition of Soar as a psychological model [31], Newell advocated a cumulative methodology in the spirit of Lakatos [25], under pressure from critics he later took a much weaker position, arguing that

> There is no essential Soar, such that if we change it we no longer have *the Soar theory*. [...] It must evolve to be a successful theory at each moment, eliminating some concepts and taking on others. But the evolution can lead to almost any transformation over time. [...] As long as each incremental change produces a viable [...] theory from the existing Soar theory, it will still and always be Soar. As in evolution, connection is historical. [32, p. 467 (Newell's emphasis)]

Even in this case which we take to be an extreme position, appropriate public criteria for theory change are required if the computational research programme is to have scientific credibility. Three such criteria, proposed by Anderson [2, pp. 42–43] in the context of his theory, ACT*, are:

(1) There is a broad range of evidence from numerous empirical domains that supports the change.
(2) There is no obvious way to incorporate empirical phenomena within existing mechanisms.
(3) There is a good argument for the adaptive value of the proposed mechanisms.

We would advocate a fourth:

(4) The alterations result in an increase in the theory's criticality/sensitivity ratio.

Without such criteria, there is a danger that "assumptions will be spun out in a totally unprincipled and frivolous way" [2, p. 42].

The methodology we are developing represents an advance on experimental programming in the sense which we criticised earlier—it is based on well-defined, publicly reviewable specifications of models and rigorously conducted, replicable computational experiments.

6. Conclusions

Mature methodologies for laboratory investigation need to be matched by equally well conceived methodologies for constructing and testing computational theories. Current methods of building and evaluating computational models of cognition are inadequate, particularly if we are entering a period of theoretical unification. As a step towards bridging the gap between laboratory investigation and computational theory we have argued that it is important to develop executable specification languages. One such language is Sceptic. Techniques for specifying models in this language have been developed using, as a test case, Soar, one of the most complex and challenging computational models in the literature. The results suggest that the specification approach is practical for reconstructing the Soar model, and that such models may be clarified by clearly separating essential elements of such theories from implementation details. At only 28 rules, the abstract description of Soar approaches three orders of magnitude smaller than a conventional implementation.

A great deal more work will be needed to realise a comprehensive methodology for specifying and testing computational theories, including the development of more sophisticated software tools than Sceptic. A full paradigm for cognitive modelling must address two main sets of requirements; the provision of appropriate tools for individual modelling experiments, and techniques for supporting research programmes which subsume a number of such experiments and the progressive development of theories to account for the results. In this paper we have only addressed the first set of requirements, focusing on the use of improved model specification tools. We have shown that, at a minimum, there is substantial scope for improvement in current modelling techniques.

Appendix A. A rational reconstruction of Soar 4

To illustrate the use of Sceptic this appendix provides details of a Sceptic specification of Soar 4. In order to place the specification in context, we precede it with a brief summary of the Soar architecture. The specification itself is divided into four sections—the top-level loop, the elaboration phase, the decision phase, and chunking—and each is developed in sufficient detail to capture the principle features of the published theory. The complete A specification as given here consists of 28 rewrite rules. A full set of below-the-line functions has been implemented in Prolog, with the behaviour of the resulting program corresponding exactly to that of the LISP version of Soar 4 running on published examples such as the monkey and bananas problem and the eight puzzle. A listing of the full program is available on request.

A.1. A summary of the Soar architecture

A.1.1. Overview of processing

Soar is an architecture, a fixed information processing structure which is specialised to adaptively raise and resolve goals by exploiting a variable body of knowledge. It is based upon the well-established approach to modelling problem solving as search in a problem space [34]. Soar implements its search by cyclically retrieving knowledge relevant to its current goal (the elaboration phase) and making decisions about the results (the decision phase). Loosely, decisions are made about the knowledge to use in resolving goals (the problem space to employ), the interpretations to assign to data (the state of that problem space), and actions to carry out (the operator to apply to the current state). If Soar has sufficient knowledge in a particular context the goal will be solved directly. If its knowledge does not embody a solution the architecture will impasse; this means that it will automatically generate a new, impasse-specific goal and recursively attempt to solve it. If the subgoal succeeds, the impasse is resolved and the result becomes available in the parent goal context. During processing of an impasse the architecture is able to identify critical contextual data and chunk this as new fragments of knowledge for future use.

A.1.2. Memories and structures

The Soar architecture makes the traditional distinction between a transitory or short term memory referred to as working memory, and a permanent or long term memory referred to as recognition memory. Though the distinction is traditional, it is difficult to equate either memory with instances of similar concepts in the psychological literature. Working memory, for example, bears little relation to how the concept is currently used in the psychological literature (e.g., [3]).

The contents of working memory represent all known aspects of the current problem solving context in the form of augmentations, or attribute/value pairs. Augmentations describe problem spaces, which are, in effect, the representation in which problem solving is or could be carried out, states (features of the current problem solving context) and operators (external or internal actions which modify the current state). For example if the problem space attribute of goal G14 has P2 as its value this is represented as the augmentation structure `(goal G14 ^problem-space P2)`. If the name of the problem space P2 is `eight-puzzle` then this is represented by the augmentation `(problem-space P2 ^name eight-puzzle)`.

Also contained in working memory are preferences. These encode the relative or absolute worth of current and potential augmentations. Preferences are primarily used in selecting among competing problem spaces, states and operators. A preference may indicate, for example, that within the current goal one operator is better than some other operator. In Soar 4 preferences and augmentations are stored together in a single working memory.

Recognition memory is closely related to the rule memory of earlier production system architectures, and consists of an unstructured set of productions. These represent long term knowledge of co-occurrence relations between working memory elements. Productions consist of a set of "matches" (templates which must be matched against elements in working memory for the production to be applicable) and "makes" (which indicate the working memory elements that should be created if the production applies). Variables can be shared between template matches and makes, in which case they must be consistently bound on any application of the production. Applying a production amounts to retrieval of information from long term memory.

A.1.3. The processing cycle

The Soar architecture operates in a cyclic fashion, with each cycle consisting of an elaboration phase followed by a decision phase.

The elaboration phase

The elaboration phase is itself cyclic. Each cycle consists of an input cycle in which input/perceptual channels are checked and the results added to working memory, an elaboration cycle in which productions are fired (the right-hand sides of applicable productions are added to working memory), and an output cycle in which output is generated from various designated motor augmentations. The perceptual input and motor output cycles are not theoretically well advanced, and we shall not consider these in detail here. It is primarily differences in the elaboration cycle which differentiate Soar 4 from later versions, though in all versions the purpose of the elaboration cycle is two-fold: to elaborate the representation of objects in working memory by further augmenting those objects; and to create preferences for future possible problem spaces, states, and operators. The same basic mechanism, the firing of productions, achieves both of these purposes.

A production may be fired if there is a mapping between existing working memory elements and the elements of the left-hand side of that production such that all variables in the production's conditions are consistently mapped. Each consistent mapping yields an instantiation of the production.

One elaboration cycle of Soar 4 consists of firing all production instantiations in parallel which are licensed by the contents of working memory but which have not previously been fired. This generally results in augmentations and preferences being added to working memory. The instantiation is also recorded to prevent subsequent identical firings.

The alterations to working memory caused by each elaboration cycle and subsequent input cycle may result in new production instantiations being licensed. These instantiations are fired in the next elaboration cycle. Cycling within the elaboration phase continues until no further instantiations are licensed. At this point, termed quiescence, the elaboration phase terminates.

The decision phase

The decision phase results in either the selection of a context object (a problem space, state, or operator) or the creation of a subgoal in response to an impasse. If a context object can be selected based on the preferences available at the beginning of the relevant decision cycle, then working memory is modified by replacing the previous value of the object with the selected value. If no object can be selected then Soar automatically creates a subgoal to resolve the impasse. Soar has general, default knowledge encoded in a set of default productions which is used for resolving impasses and thereby permitting problem solving to proceed.

Once a goal has been raised a problem space must be selected. The choice of problem space determines the possible states that can arise when solving a problem. In principle it might be possible to solve a problem in any of a number of representations. Correspondingly, operators apply to states and so a state must be selected before an operator can be selected. In general, once a problem space and initial state have been selected, operators are selected and applied in successive decision cycles until Soar detects that the current goal has been achieved. At this point Soar terminates any outstanding subgoals of the achieved goal which are now irrelevant thus restoring the context in which the goal originally arose.

A.1.4. Learning

Soar also includes an experience-based learning mechanism, chunking, which can be triggered by the firing of any production. Each production fires in the context of a particular goal. When productions fire, any elements created which augment supergoals of their firing goal are detected. These elements are used to construct new productions (chunks) which are then added to recognition memory. Once a chunk has been created it behaves as any other production, so if Soar finds itself attempting to solve a goal similar to that which previously lead to a chunk, the chunk will fire, possibly avoiding the need for subgoaling in this new situation. The left-hand side of the chunk is built from the elements relating to the supergoal and possibly higher goals which contributed to the creation of the subgoal results. The right-hand side of the chunk is built directly from the results.

A.2. A Sceptic specification of Soar 4

A.2.1. The Soar decision cycle

Overview of the decision cycle

As described above, top-level processing comprises repeated decision cycles. Each decision cycle consists of an elaboration phase where all long term knowledge is brought to bear followed by a decision phase where a decision based on the current preferences is effected.

Above-the-line specification of the decision cycle

The Soar top-level cycle is captured by two rules, one which effects a single decision cycle (consisting of an elaboration phase followed by a decision phase), and one which effects repeated decision cycles (by triggering one `decision_cycle` before triggering itself). These rules could be merged into a single rule, but the division distinguishes between the cycling and the processing within each cycle.

```
soar:
    true
 => decision_cycle,
    soar.

decision_cycle:
    true
 => elaboration_phase,
    decision_phase.
```

On the occurrence of the trigger `soar` Sceptic checks the conditions of all relevant rules. There is just one rule, and since its conditions are defined to be unconditionally true, the triggers on the right-hand side of the rule are issued in sequence, first triggering a single decision cycle and then recursively triggering `soar`. On each decision cycle, the two triggers on the right-hand side of the second rule, corresponding to the elaboration phase and decision phase described above, are called in sequence. Once initiated, the `soar` process cycles indefinitely. There is an obvious difference here between the Sceptic specification and the LISP implementations, which allow processing to be interrupted. Whilst this is obviously a very useful feature of the implementations, it is the first of several clear differences between the implementation and the cognitive theory.

Below-the-line processes in the decision cycle

The only state tester used in the above specification is `true`. No generators are employed. The two right-hand side triggers not defined above, `elaboration_phase` and `decision_phase`, can, at a highly abstract level, be classed as primitive updating functions. That is, this specification of the top level of Soar may be viewed as an extremely coarse specification of the entire architecture, with `elaboration_phase` and `decision_phase` being interpreted as primitive actions.

Summary and discussion of decision cycle

Although the specification consists of only two rules it is a substantive summary of the Soar processing structure. It specifies, in a formal language which is executable, the main subprocesses involved in the Soar architecture, and the way in which those processes fit together.

Of course, the description is very different from the descriptions of Newell and others discussed in Section 2.1. It abstracts away a massive amount of detail: much of the theory lies in the primitives `elaboration_phase` and `decision_phase` which are defined below the line. It is the purpose of the remaining specifications in this appendix to put this detail back, but to do so in such a way that we only specify the essential details. For example the LISP implementations have various mechanisms to interrupt and trace the problem solving. From the perspective of specifying the cognitive theory, it is important that this specification does not include such details, no matter how obvious it is that they are only for implementational purposes, since one cannot be sure that the process of providing them may not substantively influence the implementation, the system's behaviour, or even the way theorists interpret the theory.

A.2.2. Elaboration phase processes

Overview of the elaboration phase

The Soar architecture takes in information from its environment and stores it in working memory. Relevant knowledge is then retrieved by firing productions. This results in elements being added to working memory in the elaboration cycle. Finally, distinguished working memory elements are interpreted as motor commands. This process is repeated until quiescence.

Above-the-line specification of the elaboration phase

The elaboration phase consists of eight rules. The first of these effects initialisation and an initial input cycle before triggering the cycling of the phase.

```
elaboration_phase:
    true
 => initialise_elaboration_phase,
    input_cycle,
    cycle_if_not_quiescent.
```

Initialisation consists of marking all currently existing working memory elements. This is necessary because the Soar decision procedure distinguishes between elements created in the most recent elaboration phase and older working memory elements.

```
initialise_elaboration_phase:
    wm_match(WME)
 => wm_mark(WME).
```

The rule `cycle_if_not_quiescent` effects cycling within the elaboration phase. The antecedent condition tests whether the architecture has become quiescent or not. If Soar is quiescent then the trigger will rewrite as the empty sequence of actions, terminating the elaboration phase and returning control to the top-level rule which triggers the next top-level phase, the decision phase. If quiescence has not been reached then an elaboration cycle will be triggered, followed by an output cycle and a further input cycle. The rule will then trigger itself recursively, effecting further elaboration, output and input, until quiescence is reached and the rule's condition fails.

```
cycle_if_not_quiescent:
    not(quiescent)
 => elaboration_cycle,
    output_cycle,
    input_cycle,
    cycle_if_not_quiescent.
```

Neither the input cycle nor output cycle are well described in the Soar literature. It is clear, however, that the input cycle adds elements to working memory, whereas the output cycle effects motor commands based on the contents of working memory. The specification given here also assumes that the output cycle removes output commands from working memory when those commands have been carried out.

```
input_cycle:
    generate_input_wme(WME, Justification)
 => wm_make(WME, Justification).

output_cycle:
    match_output_wme(WME, Command)
 => effect_motor_command(Command),
    wm_remove(WME).
```

The elaboration cycle is specified by a cascade of three rules in which information is passed between rules as parameters associated with the rule triggers. The `elaboration_cycle` rule finds all instantiations of productions whose conditions are satisfied by the current contents of working memory, and which have not already been fired. Recall that variables in Sceptic conditions are universally quantified, so this rule is triggered once for each distinct instantiation. The generator `generate_unrefracted_instantiation` also generates a firing goal for each instantiation. This is the lowest goal matched in the firing of the production, and it is used when checking that the instantiation has not previously been fired (via `im_match`), when creating an instantiation memory element for the particular firing (`im_make`), and in chunking which is triggered by `fire_production`. The right-hand side of the rule consists of two triggers: `im_make`, an updater which makes an element in instantiation memory, effectively marking the instantiation to prevent future firings of the same instantiation, and `fire_production`, a rule which creates the appropriate working memory elements as specified by the right-hand side of the firing production. The `create_elements` rule adds augmentations and preferences to working memory together with their justifications. Justifications are not used for truth maintenance in Soar 4 (as they are in Soar 5), but they are used by the learning mechanism.

```
elaboration_cycle:
    generate_unrefracted_instantiation(Instantiation,
                                       FiringGoal),
    not(im_match(Instantiation, FiringGoal))
 => im_make(Instantiation, FiringGoal),
    fire_production(Instantiation, FiringGoal).
fire_production(Instantiation, FiringGoal):
    generate_elements_to_make(Instantiation, ElementsToMake),
    generate_justification(Instantiation, Justification)
 => create_elements(ElementsToMake, Justification),
    build_chunks(FiringGoal, ElementsToMake).

create_elements(ElementsToMake, Justification):
    member(WME, ElementsToMake)
 => wm_make(WME, Justification).
```

The build_chunks rule is discussed separately in Section A.2.4. Note that this process must follow the creation of working memory elements—it cannot be a parallel process—as it relies upon those elements and their justifications being in memory.

Below-the-line processes in the elaboration phase

Two state testers are required in the elaboration phase: quiescent, which succeeds when there are no valid production instantiations which have not been fired (i.e., which have not already been recorded in instantiation memory); and im_match which checks that an instantiation is currently in instantiation memory. This second state tester could also be used to generate the elements of instantiation memory. It acts as a state tester here because, when it is called, its arguments will be instantiated.

The elaboration phase requires generators to identify and instantiate productions (together with their firing goal) which are applicable during the current elaboration phase (generate_unrefracted_instantiation) and to identify the elements (and the justification for those elements) which should actually be created by a production firing (generate_elements_to_make and generate_justification). Generators are also used by initialise_elaboration_phase, input_cycle, output_cycle, and create_elements, which uses member, a standard library generator.

The principle updater used during the elaboration phase is wm_make. This adds an element and its justification to working memory if it is not already present. Several other updaters are also employed: wm_mark marks an existing working memory element, indicating that it existed prior to the current elaboration phase; wm_remove removes an element from working memory; im_make adds an element and its firing goal to instantiation memory; and effect_motor_command is a primitive action intended to serve as the interface to the motor system.

Summary and discussion of elaboration phase

The essential structure of the elaboration phase is captured by eight Sceptic rules. They describe the sequencing of the subprocesses; the creation of working memory elements during the input cycle; the firing of all matching but previously unfired productions, and the execution of motor commands in the output cycle. As an above-the-line specification of all essential processes in the elaboration phase it is complete except for the chunk building process. This process, which is triggered by the rule which is responsible for firing productions, is discussed in more detail below.

The elaboration phase also requires a number of below-the-line functions; six specialised functions for testing the processing state and generating results from this, and six primitive functions for updating the state. Several of these are also used in other phases of processing.

The modest size of this specification makes it relatively easy to understand the essential mechanisms of elaboration involved in Soar. The below-the-line functions, which we have not described in detail, involve significant programming (14 Prolog procedures) but are of less importance to researchers wishing to get to grips with the essential Soar theory, to formulate predictions from it, or to design alternative mechanisms.

A.2.3. The decision phase

Overview of decision phase

The decision phase attempts to select a problem space, state or operator for the current context. If this selection is successful then the selection is installed by adjusting the context stack appropriately. If the selection fails an impasse arises and a subgoal to resolve it is generated.

A further aspect of the decision phase not generally discussed concerns the adjustment of item augmentations. Impasses that arise when several competing objects are equally acceptable for a context slot include in their representation an augmentation of the relevant goal for each competing object. These augmentations are referred to as item augmentations. Preferences created during an elaboration phase may contradict current item augmentations, or suggest further item augmentations, for any goal arising from such an impasse. Consequently, after each elaboration phase, any item augmentations of a goal that are no longer applicable should be removed and item augmentations should be created for any new objects competing in the impasse. In the LISP implementations, this process is intertwined with the other algorithms effecting the decision phase, but the process is independent, and treating it as such clarifies the specification and the nature of item augmentation adjustment.

Above-the-line specification of decision phase

The decision phase can be viewed as having three parts. One is a cascade of operations which identify candidate values for the context object, select and install the preferred value, and terminate any subgoals which are no longer relevant. This relatively simple structure is complicated by the possibility that impasses may arise during the operation, so a second set of processes is required to process the impasses and create the corresponding subgoals. The distinct process of item augmentation adjustment forms a third part of the decision phase.

If a context slot needs to be (re)decided and this can be determined by looking at the newly created preferences in working memory, then the selection process is run for that slot. If there is no context slot that needs to be decided then there is a no-change impasse. This will occur if, for example, all problem spaces, states and operators currently have values and no preferences where generated on the previous elaboration phase which should alter those values.

```
decision_phase:
    generate_context_pair_to_alter(Goal, Attribute)
 => attempt_selection_of_new_object(Goal, Attribute).
decision_phase:
    not(generate_context_pair_to_alter(_Goal, _Attribute))
 => no_change_impasse.
```

Given that a decision is required for a particular context slot, an object can be chosen randomly from the set of possible objects that may, according to the current preferences, fill that slot, but only provided that all of those possible values are "mutually indifferent" (which may be the case if, for example, the set is empty or if several values have conflicting preferences). If the set of possible values is not mutually indifferent then a context slot impasse arises.

```
attempt_selection_of_new_object(Goal, Attribute):
    generate_possible_values_for_attribute(Goal, Attribute,
                                           ValueSet),
    mutually_indifferent(Goal, Attribute, ValueSet)
 => select_and_install_object(Goal, Attribute, ValueSet).
attempt_selection_of_new_object(Goal, Attribute):
    generate_possible_values_for_attribute(Goal, Attribute,
                                           ValueSet),
    not(mutually_indifferent(Goal, Attribute, ValueSet))
 => context_slot_impasse(Goal, Attribute, ValueSet).
```

The selection and installation process consists of three independent subprocesses:
(1) a value is selected from among those possible and installed in the appropriate slot;
(2) all context slots within the current goal following the slot on which the decision is being made are reinitialised; and
(3) all goals below the goal which the selected object augments are terminated.

```
select_and_install_object(Goal, Attribute, ValueSet):
    random_select(ValueSet, Value)
 => install_context_object(Goal, Attribute, Value),
    reinitialise_following_attributes(Goal, Attribute),
    terminate_subgoals(Goal).

install_context_object(Goal, Attribute, Value):
    responsible_preferences(augmentation(goal, Goal, Att=Val),
                            Conds)
 => wm_replace(augmentation(goal, Goal, Att=Val), Conds).

reinitialise_following_attributes(Goal, Attribute):
    following_context_attribute(Attribute,
                                AttributeToInitialise)
 => initialise_attribute(Goal, AttributeToInitialise).
```

Subgoal termination involves removing all working memory elements which are not linked to the goal stack above the terminated goal and removing all elements from instantiation memory which are no longer relevant. These are all elements whose firing goal has been terminated.

```
terminate_subgoals(Goal):
    generate_subgoal_wme(WME, Goal)
 => wm_remove(WME).
terminate_subgoals(Goal):
    generate_subgoal_ime(Goal, Instantiation, FiringGoal)
 => im_remove(Instantiation, FiringGoal).
```

Turning to impasse processing, when a context slot impasse occurs, all previous subgoals of the goal on which the impasse arose are terminated and a new subgoal is created. Appropriate architecture-generated augmentations are added to that subgoal. The attribute in the context slot that caused the impasse, as well as all lower attributes within that context, are reinitialised. Creating a subgoal requires a new identifier to be generated and several impasse-specific augmentations to be added to working memory.

```
context_slot_impasse(Goal, Attribute, ValueSet):
    detect_impasse_type(Goal, Attribute, ValueSet, SubType)
 => terminate_subgoals(Goal),
    create_subgoal(Goal, context_slot-SubType,
                   Attribute, ValueSet).
context_slot_impasse(Goal, Attribute, _ValueSet):
    generate_attribute_to_initialise(Attribute,
                                     AttributeToInitialise)
 => initialise_attribute(Goal, AttributeToInitialise).

create_subgoal(Goal, Type, Attribute, ValueSet):
    generate_goal_identifier(SubGoal)
 => add_subgoal_augmentations(Goal, SubGoal, Type,
                              Attribute, ValueSet).

add_subgoal_augmentations(SuperGoal, Goal, Type,
                          Attribute, ValueSet):
    generate_impasse_augmentation(SuperGoal, Goal, Type,
                                  Attribute, ValueSet,
                                  Att=Val, Justification)
 => wm_make(augmentation(goal, Goal, Att=Val), Justification).
```

The processing required for a no-change impasse is similar except that, because no-change impasses can only occur on the lowest context slot of the lowest goal, no subgoals need be terminated and no context slots need be reinitialised. Instead a subgoal of the lowest goal is created, again with appropriate architecture-generated augmentations.

```
no_change_impasse:
    generate_lowest_goal(Goal)
 => create_subgoal(Goal, no_change, _, _).
```

The Sceptic reconstruction highlights two distinct types of impasse: those arising from situations in which no context slot should be altered (no-change impasses), and those arising from situations where a choice is required, but no choice can be made (context slot impasses). The different types of impasse arise at different stages of processing, and in response to different failures of problem solving. However, the distinction, whilst often discussed within the Soar community at the level of problem spaces, states and operators, rather than the level of individual Soar productions, is not generally regarded as a distinction made within the architecture. The Sceptic reconstruction demonstrates that genuine processing differences underlie the distinction.

The basic structure of the item adjustment step is straightforward. This is captured by a pair of rules, one of which is responsible for removing invalid augmentations and one for creating new ones. Both of these rules are triggered by `decision_phase`, corresponding to the fact that this process, which can be treated independently of the rest of the decision phase, is triggered when a decision phase is triggered.

```
decision_phase:
    wm_match(augmentation(goal, Goal, item=Item)),
    not(generate_item_augmentation(Goal, Item, Justification))
 => wm_remove(augmentation(goal, Goal, item=Item)).
decision_phase:
    generate_item_augmentation(Goal, Item, Justification),
    not(wm_match(augmentation(goal, Goal, item=Item)))
 => wm_make(augmentation(goal, Goal, item=Item), Justification).
```

Below-the-line specification of the decision phase

A number of special generators are required for decision phase processing. These recognise conditions where context objects need to be installed, and generate possible candidates, augmentations, justifications, subgoals and so forth. A single state tester, for determining whether alternative context objects are mutually indifferent, is also required.

General updaters are required for adding and removing elements from working memory, as well as for initialising context slots and removing elements from instantiation memory.

Item adjustment requires two left-hand side conditions. In the first rule, `wm_match` is used to generate an existing item augmentation and `generate_item_augmentation` is used to check whether this item augmentation is justified. If not, the item augmentation is removed. The second rule uses these conditions in the opposite way. `generate_item_augmentation` is used to generate an item augmentation and its justification and if that augmentation is not present in working memory then it is added.

The only updating mechanisms required during item adjustment are the standard operations for adding and deleting items from working memory (`wm_make` and `wm_remove`).

Summary and discussion

Decision phase processing is the most complex part of the Soar architecture, and this is reflected in the relative complexity of the Sceptic specification, which consists of 14 rules and 12 specialised generators. Again, the generators require significant coding, but the complexity of this coding is not relevant at the required level of abstraction of the Soar cognitive theory.

Item adjustment is conceptually a simple mechanism for ensuring that current augmentations are consistent and complete. This simplicity is reflected in the brevity of the above-the-line specification. This specification does suggest, however, that item adjustment can be seen as a restricted form of truth maintenance, similar to those truth maintenance systems developed in AI [26] and explicitly employed in later versions of Soar, though for a different purpose.

A.2.4. Chunking

Overview of chunking

Chunks are created whenever a production which fired within the context of the most recent subgoal adds elements to a context associated with a supergoal. The right-hand side of the chunk is derived from those elements added to the supercontext, with the left-hand side being determined by recursively tracing the justifications of the elements which triggered the production to elements in supercontexts.

Above-the-line description of chunking

The above-the-line specification of chunking consists of just two rules. The first rule attempts to build the appropriate chunk given a production's firing goal and results.

```
build_chunks(FiringGoal, Results):
    generate_lowest_goal(FiringGoal),
    new_supergoal_elements(Results, FiringGoal, ElementsToMake),
    conditions_prior_to_goal(ElementsToMake, FiringGoal,
                             ElementsToMatch),
    assemble_chunk(ElementsToMatch, ElementsToMake,
                   InstantiatedChunk),
    variablise_production(InstantiatedChunk, VariablisedChunk)
 => add_chunk(VariablisedChunk, InstantiatedChunk).
```

The second rule only fires if the potential chunk generated by the first rule does not duplicate any previously learned chunk. If this rule does fire, it adds the new chunk to recognition memory and refracts it by adding the appropriate instantiation to instantiation memory, hence preventing it from firing immediately with the instantiation which led to its creation.

```
add_chunk(VariablisedChunk, InstantiatedChunk):
    not(test_duplicate_chunk(InstantiatedChunk)),
    generate_firing_goal(InstantiatedChunk, FiringGoal)
 => im_make(InstantiatedChunk, FiringGoal),
    rm_make(VariablisedChunk).
```

These two rules could be merged into a single rule, but the separation appears to be a natural one. The first rule essentially generates a potential new element of recognition memory, and the second rule records this element if it does not duplicate an existing element.

Below-the-line description of chunking

A single state tester in the first rule ensures that chunking only occurs when the firing goal is the most recently created subgoal. Several generators are also used by this rule to generate the chunk from the firing goal and results. The first, `new_supergoal_elements`, generates the subset of results which belong to the supercontext. The second, `conditions_prior_to_goal`, determines the working memory elements from the supercontext which indirectly led to creation of these results. The third, `assemble_chunk`, uses the results of the previous two generators to construct an instantiated version of the final chunk. Finally, the generator `variablise_chunk` generates an uninstantiated chunk from the instantiated version by replacing all identifiers with variables.

The second rule also uses a single state tester, checking that the generated chunk does not duplicate an existing production. The firing goal of the chunk is generated so that the appropriate information may be added to instantiation memory.

Summary and discussion

In its barest form chunking is very simple. As with other Soar processes, the above-the-line specification of chunking is very brief. Considerable detail is again hidden below the line, but this reveals that at this level of abstraction chunking is not a complex procedure. Nevertheless, there are several differences between this specification and the LISP implementation. Most notable of these is that in our specification chunking is restricted to the most recently created subgoal, whereas chunking in the LISP implementations is switchable between this and chunking over all goals. Newell [31] argues from psychological data that chunking in the human information processing architecture is bottom-up. All-goals chunking has therefore been disregarded here. More minor details present in the LISP implementations of chunking but not in this specification include: partitioning of results into linked subsets, with each such subset leading to distinct chunks; expansion of each subset of results to include all elements linked via some chain of working memory elements to one of the results; removal of condition matches which add little constraint to the potential matching of the chunk; addition of inequality restrictions to matches within the chunk; splitting of operator application chunks into subchunks which separately apply the operator and create the new state; and the filtering of potential chunks generated in certain problem spaces. The omission of these aspects is justified by their dubious status as part of the Soar cognitive theory—while they are described in the manuals which accompany the Soar implementations [22,24], they are not mentioned in cognitive psychological descriptions of Soar [23,31,40].

References

[1] AAAI, *Uncertainty in Artificial Intelligence* **1–7** (1986)–(1992).

[2] J.R. Anderson, *The Architecture of Cognition* (Harvard University Press, Cambridge, MA, 1983).
[3] A.D. Baddeley, *Working Memory* (Oxford University Press, Oxford, 1986).
[4] P.J. Barnard, Interacting cognitive subsystems: a psycholinguistic approach to short-term memory, in: A. Ellis, ed., *Progress in the Psychology of Language* (Lawrence Erlbaum, Hillsdale, NJ, 1985) Chapter 6, 197-258.
[5] B. Carpenter, *The Logic of Typed Feature Structures*, Cambridge Tracts in Theoretical Computer Science **32** (Cambridge University Press, Cambridge, UK, 1992).
[6] W. Clocksin and C. Mellish, *Programming in Prolog* (Springer, New York, 1981).
[7] R. Cooper, A sceptic specification of Johnson-Laird's "Mental Models" theory of syllogistic reasoning, Tech. Rept. UCL-PSY-ADREM-TR4, Department of Psychology, University College London, London (1992) 9 pp.
[8] R. Cooper, A sceptic specification of Sloman's motive processing engine and its application in the nursemaid scenario, Tech. Rept. UCL-PSY-ADREM-TR3, Department of Psychology, University College London, London (1992) 15 pp.
[9] R. Cooper and J. Farringdon, Sceptic version 4 user manual, Tech. Rept. UCL-PSY-ADREM-TR6, Department of Psychology, University College London, London (1993).
[10] R. Cooper and T. Shallice, Soar and the case for unified theories of cognition, *Cognition* **55** (1995) 115-149.
[11] R. Cooper, T. Shallice and J. Farringdon, Symbolic and continuous processes in the automatic selection of actions, in: J. Hallam, ed., *Hybrid Problems, Hybrid Solutions*, Frontiers in Artificial Intelligence and Applications **27** (IOS Press, Amsterdam, 1995) 27-37.
[12] E. Davis, *Representations of Common Sense Knowledge* (Morgan Kaufmann, San Mateo, CA, 1990).
[13] J. Farringdon, Translating a back-propagation network into propositional calculus, and reasoning with such translations over time, Master's Thesis, South Bank Polytechnic, London (1990).
[14] J. Fox, Techniques for developing distributed decision systems, Dilemma Workshop # 1 (1992).
[15] M.R. Genesereth and N.J. Nilsson, *Logical Foundations of Artificial Intelligence* (Morgan Kaufmann, Los Altos, CA, 1987).
[16] S. Hajnal, J. Fox and P. Krause, Sceptic user manual: version 3.0, Tech. Rept., Advanced Computation Laboratory, Imperial Cancer Research Fund, London (1990).
[17] G.E. Hinton and T. Shallice, Lesioning an attractor network: investigations of acquired dyslexia, *Psych. Rev.* **98** (1991) 74-95.
[18] E. Hunt and R.D. Luce, Soar as a world view, not a theory, *Behav. Brain Sci.* **15** (1992) 447-448.
[19] M. Johnson, *Attribute-Value Logic and the Theory of Grammar*, CSLI Lecture Note Series **16** (CSLI, Stanford, CA, 1988).
[20] P.N. Johnson-Laird, *Mental Models* (Cambridge University Press, Cambridge, UK, 1983).
[21] R. Kowalski, *Logic for Problem Solving* (North-Holland, New York, 1979).
[22] J.E. Laird, C.B. Congdon, E. Altmann and K. Swedlow, Soar user's manual: Version 5.2, Tech. Rept. CMU-CS-90-179, Carnegie Mellon University, Pittsburgh, PA (1990).
[23] J.E. Laird, A. Newell and P.S. Rosenbloom, SOAR: an architecture for general intelligence, *Artif. Intell.* **33** (1987) 1-64.
[24] J.E. Laird, K. Swedlow, E. Altmann, C.B. Congdon and M. Weismeyer, Soar user's manual: Version 4.5, Tech. Rept., University of Michigan, Ann Arbor, MI (1989).
[25] I. Lakatos, Falsification and the methodology of scientific research programmes, in: I. Lakatos and A. Musgrave, eds., *Criticism and the Growth of Knowledge* (Cambridge University Press, Cambridge, UK, 1970) 91-196.
[26] J.P. Martins, The truth, the whole truth and nothing but the truth, *AI Magazine* **11** (5) (1991) 7-25.
[27] B.G. Milnes, A specification of the Soar cognitive architecture in Z, Tech. Rept. CMU-CS-92-169, Carnegie Mellon University, Pittsburgh, PA (1992).
[28] J. Morton, R.H. Hammersley and D.A. Bekerian, Headed records: a model for memory and its failures, *Cognition* **20** (1985) 1-23.
[29] J. Morton and K.E. Patterson, A new attempt at an interpretation, or, an attempt at a new interpretation, in: *Deep Dyslexia* (Routledge, London, 1980) 91-118.
[30] A. Newell, You can't play 20 questions with nature and win, in: W.G. Chase, ed., *Visual Information Processing* (Academic Press, San Diego, CA, 1973) 283-308.

[31] A. Newell, *Unified Theories of Cognition* (Harvard University Press, Cambridge, MA, 1990).

[32] A. Newell, Soar as a unified theory of cognition: issues and explanation, *Behav. Brain Sci.* **15** (1992) 464–492.

[33] A. Newell, J.C. Shaw and H.A. Simon, Elements of a theory of human problem solving, *Psych. Rev.* **65** (1958) 151–166.

[34] A. Newell and H.A. Simon, *Human Problem Solving* (Prentice Hall, Englewood Cliffs, NJ, 1972).

[35] D.A. Norman and T. Shallice, Attention to action: willed and automatic control of behavior, in: R. Davidson, G. Schwartz and D. Shapiro, eds., *Consciousness and Self Regulation: Advances in Research and Theory, Vol. 4* (Plenum, New York, 1986) 1–18.

[36] M. O'Neill, The STOP demonstrator, Tech. Rept., Advanced Computation Laboratory, Imperial Cancer Research Fund, London (1992).

[37] D. Partridge, *Engineering Artificial Intelligence Software* (Intellect Books, Oxford, 1992).

[38] D. Plaut and T. Shallice, Deep dyslexia: a case study of connectionist neuropsychology, *Cognit. Neuropsych.* **10** (1993) 377–500

[39] P. Rabbitt, Does it all go together when it goes?, *Quart. J. Experim. Psych.* **46A** (1993) 385–434.

[40] P. Rosenbloom, J. Laird, A. Newell and R. McCarl, A preliminary analysis of the Soar architecture as a basis for general intelligence, *Artif. Intell.* **47** (1991) 289–325.

[41] T.A. Salthouse, Effects of age on verbal abilities: an examination of the psychometric literature, in: L.L. Light and D.M. Burke, eds., *Language and Memory in Old Age* (Cambridge University Press, Cambridge, 1988) 17–35.

[42] T. Shallice, *From Neuropsychology to Mental Structure* (Cambridge University Press, Cambridge, 1988).

[43] A. Sloman, Motives, mechanisms, and emotions, *Cognit. Emotion* **1** (1987) 217–233

[44] J.M. Spivey, *The Z Notation: A Reference Manual*, Prentice Hall International Series in Computer Science (Prentice Hall, Englewood Cliffs, NJ, 1989).

[45] F. van Harmelen and J. Balder, $(ML)^2$: a formal language for KADS models of expertise, *Knowl. Acquisit.* **4** (1992) 127–161.

[46] N.S. Walker, Biomedical image interpretation, Ph.D Thesis, Department of Computer Science, Queen Mary and Westfield College, London (1991)

[47] B. Wielinga, A. Schreiber and J. Breuker, KADS: a modelling approach to knowledge engineering, *Knowl. Acquisit.* **4** (1992) 5–53.

[48] D. Zicha and J. Fox, Symbolic simulation of complex biological processes, Tech. Rept., Advanced Computation Laboratory, Imperial Cancer Research Fund, London (1991).