Towards an Object-Oriented Language for Cognitive Modeling

Richard Cooper
Department of Psychology
University College London
Gower Street
London WC1E 6BT
r.cooper@psychol.ucl.ac.uk

Abstract

This paper describes work towards an object-oriented language for cognitive modeling. Existing modeling languages (such as C, LISP and Prolog) tend to be far removed from the techniques employed by psychologists in developing their theories. In addition, they encourage the confusion of implementation detail necessary for computational completeness with theoretically motivated aspects. The language described here (OOS) has been designed so as to facilitate this theory/implementation separation, while at the same time simplifying the modeling process for computationally non-sophisticated users by providing a set of classes of basic "cognitive" objects. The object classes are tailored to the implementation of functionally modular cognitive models in the box/arrow style. The language is described (in terms of its execution model and its basic classes) before a sketch is given of a simple production system which has been implemented within the language. We conclude with a discussion of on-going work aimed at extending the coverage of the language and further simplifying the modeling process.

Introduction: Rationale

The principle of "functional modularity", whereby the behavior of a complete system is determined by the interaction of a number of semi-autonomous sub-systems, is a commonplace within cognitive science. Many cognitive models, including those from both the connectionist paradigm and the symbolic paradigm, are based on the principle. The former is exemplified by models such as Miikkulainen's model of script processing (Miikkulainen, 1993) and Burgess & Hitch's model of the articulatory loop (Burgess & Hitch, 1992). The latter is exemplified by models such as Barnard's Interacting Cognitive Subsystems model (Barnard, 1985) and production systems such as Soar (Newell, 1990, which is modular in the sense of having separable working memory and production memory components). Hybrid symbolic/connectionist models also employ functional modularity (e.g., Wermter & Lehnert, 1989), and further models, such as Morton's model of the processing of words and pictures (Morton, 1981), adopt functional modularity without making any commitment to the underlying implementation.

Within cognitive psychology, functional modularity is often expressed in terms of box/arrow diagrams. These diagrams, which generally consist of a number of labeled interconnected boxes, have a long and checkered history stretching back to Lichtheim (1885). Early criticisms failed to differentiate between functional and structural modularity (see Shallice (1988) for a review). More recently, critics have argued that such diagrams are virtually contentless, but this accusation can be rebutted by observing that the diagrams do make theoretical claims about functional modularity and the flow of data between the modules. Such theoretical claims may be justified by, and tested against, empirically observed dissociations between functioning (cf. Morton, 1981; Shallice, 1988).

Box arrow diagrams share a superficial resemblance to flow charts, and this too has led to criticism. However, the two diagrammatic notations differ on several substantive grounds. Crucially, flow charts encode algorithms and express the flow of control. They are rooted in the traditional model of computation as sequential processing, and (if taken to be more than purely descriptive of behavior) seem to imply that flow charts are somehow represented in the head and executed by some conventional computational device. Box/arrow diagrams, in contrast, express functional modularity and flow of data between functional modules. They make no claims about sequential processing, and do not suggest the existence of any program which the system deliberately follows.

Independently of their use in cognitive psychology, ideas similar to those behind functional modularity have recently achieved prominence within computer science. In particular, the object-oriented paradigm (see, e.g., Rumbaugh *et al.*, 1991) advocates the use of informationally encapsulated objects to which sub-computations can be allocated via specified communication channels. As a result, Object-Oriented Programming (OOP) offers the possibility of directly addressing, within a sound computational framework, the functional modularity implicit in box/arrow diagrams. Specifically, within an object-oriented paradigm, boxes might be directly modeled as objects (of various classes), with arrows being directly modeled as communication channels between those objects.

OOP has considerable potential utility within the domain of cognitive modeling in virtue of the approach that it offers to functional modularity. There are further arguments, however, for OOP within the discipline. Firstly, an object-oriented class hierarchy may be used to facilitate modeling by providing a variety of object classes tailored to the requirements of cognitive modeling. The class hierarchy described in the following section, for example, includes various different forms of buffer common in psychological theorizing. By providing the psychologist with a library of such classes, he/she may implement a model without having to consider the detailed implementation of the boxes within the model. That is, the psychologist can work at the level of interacting buffers and processes, etc., rather than at the level of C, Lisp, or Prolog code. In this way, OOP can lessen the "distance" between the language of the cognitive psychologist and the language of the implementation. Secondly, OOP offers the possibility of providing a truly declarative specification of a psychological model. There are two advantages to such a specification: 1) it frees the theorist entirely from the implementation level, so the question of whether individual boxes are implemented in symbolic or connectionist terms is side-stepped, and the important issue of specifying the properties of the individual boxes comes into focus;[1] and 2) the box/arrow specifications with which cognitive psychologists work are equally declarative.

The declarative statement offered by OOP of box/arrow diagram comes about by directly mapping the diagram to a specification of object instances and communication channels. Extending that statement to a complete declarative (and executable) specification requires declarative specifications of the processes internal to each box, but such specifications may be given in, for example, the purely logical fragment of Prolog. While this does not entirely free the psychologist from traditional programming, it does dramatically reduce the extent of that programming.

OOS: Object-Oriented Sceptic

Is object-orientedness (including the provision of an appropriate class hierarchy) the only requirement for a cognitive modeling language? Certain functionality is implicated in a great many cognitive theories (such as pattern directed processing, content addressable memory retrieval and update, and the possibility of both sequential and parallel processing modes), and a case can be made for providing these common features as primitive operations in a modeling language. One language which provides these primitives (but is not object-oriented) is Sceptic (Hajnal *et al.*, 1989; Cooper & Farringdon, 1993). This modeling language (which is based on Prolog) has been successfully applied in the implementation of a number of cognitive theories, including theories of reasoning, memory, motivation, automatic control of action, and two versions of the Soar architecture (see Cooper *et al.*, 1993). For present purposes, the details of Sceptic are not important. This section describes OOS (Object-Oriented Sceptic), a language developed in order to extend Sceptic's capabilities by incorporating support for object-oriented programming.

The Execution Model

A cognitive model specified in OOS consists of a set of box declarations. These declarations specify the class of boxes (e.g., limited-capacity buffer: see below), their class-specific properties (e.g., the capacity of a buffer), and how they interact (i.e., the arrangement of arrows between those boxes, in terms of paired input and output ports). The underlying execution model of OOS, the mechanism by which a cognitive model specified in the language is animated, is cyclic. On each cycle each object (i.e., each box) operates on any data waiting at its input ports. The result of this processing depends on the class of object in question. A typical process will transform the data and send the transformed data as input to some other object, whereas a typical buffer will incorporate the data into its state. All objects effectively operate in parallel.

Central to the execution model is a data bus, which contains all data (or messages) in transit between boxes, i.e., all messages that have been sent along an arrow from one box but not yet received. In addition, each box has an internal state. The state of the entire model at time t is fully determined by the state of each box at time t together with the contents of the data bus at time t.

The behavior of a box over time is determined by two functions, a state transition function and an output function. Each box is completely specified by its initial state (i.e., its state at time $t = 0$) and these two functions.

If the state of box x at time t is denoted by s_x^t, its input denoted by i_x^t, and its state transition function by st_x, then:

$$s_x^{t+1} = st_x(s_x^t, i_x^t)$$

The output of a box is similarly a function of its input and its internal state, and consists of a multi-set of $\langle$message, box identifier$\rangle$ pairs, with the interpretation that $\langle m, x\rangle$ represents a message m bound for box x (and to be processed as input to box x during the next cycle).

Elements of the bus are also $\langle$message, box identifier$\rangle$ pairs. If we denote by B^t the state of the bus at time t, then the multi-set of messages in the bus at time t bound for box x, $r(B^t, x)$, is given by:

$$r(B^t, x) = \{m \mid \langle m, x\rangle \in B^t\}$$

The content of the bus at time $t+1$ is the union of the outputs of all boxes, given their state at time t, and the messages bound for them at that time. In symbols:

$$B^{t+1} = \biguplus_{x \in X} \{b \mid b = out_x(s_x^t, r(B^t, x))\}$$

where X is the set of all boxes which comprise the model, and $\uplus$ denotes multi-set union.

The Class Hierarchy

The class hierarchy developed to date is somewhat limited. The root class in the class hierarchy is box. It has four subclasses: buffer, process, data, and compound. More subclasses (such as network objects) are easily added, and it is anticipated that the class hierarchy will be extended as the need arises.

Buffers: These are boxes that store information but have a null output function: messages sent to a buffer may change its state, but they will not produce output. The utility of a buffer lies in the fact that its state may be read by process or compound boxes (see below).

Buffers are sensitive to three sorts of messages. A `clear` message will effectively remove all elements from the buffer to which it is sent by replacing the existing state with an empty state. A message of the form +X (where X is a Prolog term) will add X to the buffer. A message of the form -X will delete X from the buffer (provided X is currently in the buffer). A buffer may receive any number of messages on one and the same cycle. In this case, clear messages are processed before delete messages which are in turn processed before add messages. No ordering is specified on the processing of messages within a particular category (e.g., within the category

[1]This is not to say that the properties of certain classes of boxes might not be more easily implemented in one technology or another, but that box properties, rather than implementation technology, should be the issue under discussion.

of delete messages), as such ordering does not affect the final result of processing within a cycle. This is consistent with the treatment of the bus as a multi-set, with no ordering on its elements.

Buffers have various properties which alter the way they behave when they receive messages and when they are read. A binary property indicates whether a buffer can store duplicate copies of the same information, or whether duplicates should be ignored. A second property specifies the order of access (newest first; oldest first; or random) when the buffer is read. A third property specifies whether the contents of the buffer are subject to decay (and if so what form of decay). Current options include: none; decay after a specified number of cycles; and decay randomly with probability specified in terms of a half-life. These properties must be specified for all buffers.

The class buffer has two subclasses: unlimited capacity and limited capacity. Limited capacity buffers have two additional properties. The first specifies the capacity, and the second specifies the action to take when this capacity is exceeded (delete the most recent element to make room for the new element; delete the least recent element; delete a random element; or ignore the new element).

Specifying a buffer in OOS only requires that its subclass and the value of the appropriate parameters be specified. The use of classes and properties standardizes the notion of a buffer (thus potentially increasing communicability of theories), and also allows theorists to experiment with variations on a model by varying the properties (e.g., the decay characteristics) of individual subcomponents.

The intention is that the above properties and subclasses should cover the majority of forms of buffer employed in current psychological theorizing, though further application of the language may well reveal further properties or subclasses. The motivation for the current properties and subclasses comes partly from existing psychological theorizing and partly from logical possibilities that are consistent with this theorizing. For example, in Fodor and Frazier's model of sentence processing (Frazier & Fodor, 1978; Fodor & Frazier, 1980), during processing the Preliminary Phrase Packager (PPP) works on a fragment of the sentence under consideration. In psychological parlance, the PPP makes use of a limited capacity push through store. As each new word enters the store, it pushes the least recent entry out. Within OOS, the relevant box is a limited capacity buffer with duplicates but without decay, with access via the most recent element first and with the least recent element being deleted when the capacity is exceeded. Fodor and Frazier do not specify the capacity of the PPP's buffer,[2] but OOS allows one to experiment with different capacities (by simply varying the value of the corresponding property and conducting the appropriate simulations), thereby allowing an optimal capacity (and the effects of varying this capacity) to be determined.

[2]Frazier & Fodor (1978: 293) comment that the capacity of the PPP might not be defined in words, but in terms of syllables, morphemes, or even time slices. If capacity is to be measured in syllables, morphemes or time slices, then the input messages should be packaged as syllables, morphemes, or time slices, respectively. Capacity is defined strictly in terms of the elements which constitute the messages that a buffer receives.

Processes: Processes are defined to be objects which transform data according to fixed, well specified, rules. They may be viewed as boxes which map from one representation to another. The output function is defined to be independent of the internal state, which cannot be queried. (As such, the internal state is effectively redundant, and might as well be defined to be null, with the identity mapping as the state transition function.) Processes thus have no memory capabilities, and in this sense they are the complement of buffers, which have an internal state, but a null output function.

Two subclasses of process are available (triggered and autonomous: see below), but because of the variability of possible output functions, it is not currently possible to specify processes completely in terms of properties and further subclasses. The output function of a process must at present be defined via rewrite rules similar to standard Sceptic (see Cooper & Farringdon, 1993, for details). This is an undesirable aspect of the current system, as it requires some knowledge of a conventional text-based programming language.

Triggered processes are those which are activated by input messages. If they receive no input, they generate no output, but when triggered by input, they map that input according to their output function. In a sense, triggered processes are passive processes. Autonomous processes, on the other hand, are active processes: they actively find data (by, for example, reading the contents of a specified buffer) and produce output on the basis of that data. Triggered processes may also read a buffer's contents when calculating their output, but they will not attempt to produce output unless specifically triggered by an input message.

Returning to Fodor and Frazier's model of sentence processing, the PPP can be seen to also include an autonomous process which monitors the input buffer looking for phrasal constituents, packaging such constituents when they are found and sending them to the Sentence Structure Supervisor (SSS), a collection of boxes which combine the PPP's results into a complete phrase marker for a sentence.

Data: Data boxes may be used to supply input data to a model or to record output data from a model. The two subclasses of data box which serve these two functions are source and sink. Data sources are initialized with a list of Prolog terms (read from a file). On each cycle, if the source is not empty, the first element of this list is removed from the source and a copy of it is sent via any arrows to all boxes connected to the source. Data sinks accumulate output, functioning in the reverse way to data sources: on each cycle, any messages sent to a data sink are appended to the sink.

Data boxes generally do not belong to a model in the same way as other boxes do in that no psychological validity is typically ascribed to them. Data sources might be used, for example, to supply the posited results of perceptual processes to boxes performing more central cognitive functions (thus circumventing the problem of modeling perception), and data sinks might be used to record the sequential output of the cognitive process under investigation. Thus an appropriate data source for the Fodor and Frazier model may comprise a list of words, syllables, or morphemes which constitute the input to the PPP. A data sink may then be used to collect the phrase markers generated by the SSS.

Data boxes are very flexible and individual boxes or subsets of boxes may be tested in isolation by lifting those boxes out of the complete model and attaching data sources (with appropriate inputs) at all disconnected input ports and data sinks at all disconnected output ports. Thus, Fodor and Frazier's SSS may be tested/developed in isolation from the PPP (and *vice versa*) by replacing the boxes comprising the PPP with an appropriate data source that feeds directly into the SSS.

Compounds: Compound boxes are used where it is desirable to group other boxes together into a single functional module. They might be thought of as a box within which other boxes can be put, thus allowing a model to be hierarchically structured. There are no restrictions on the outputs or states of compound boxes, and they serve no computational function within the execution model. They are, however, vital to the structured top-down development of models (see below). With regard to the sentence processing model, it would seem appropriate to treat the PPP and the SSS each as compound boxes, both consisting of a network of processes and buffers.

Communication

Within OOS communication between boxes is generally specified by defining arrows from the source box to the target box. Thus, to specify that data should be feed from a data source to a process, an arrow must be defined from the data source to the process. Defining a second arrow from the data source to, for example, a buffer, will result in the data from the source being simultaneously sent from the source to both the process and the buffer. The exception to this simple means of establishing communication channels arises with messages sent from processes, which must be explicitly addressed to a named target box (or to several named target boxes). This is to allow a single process to generate multiple messages for a variety of target boxes. Explicit addressing is performed in modified Sceptic within the specification of a process' output function.

Methodology: Building Models with OOS

There are three stages to developing an OOS model. Firstly, the model should be drawn in diagrammatic (box/arrow) form. Where modules with both processing and buffering capabilities are required, compound boxes should be used. These can "opened up" at stage 3. Compound boxes thus facilitate a top down, structured, approach to the development of an OOS model: global characteristics of the model can be specified before lower-level details of individual compounds are considered. This is consistent with psychological theorizing, where many boxes typically have complex processing characteristics (though those processing characteristics are often only specified informally).

Secondly, the class (or subclass) of each box must be determined, and the relevant properties specified. At this stage, the capacity or decay characteristics of buffers should be specified, and questions of whether processes are triggered or autonomous must be addressed.

Thirdly, the internals of those boxes which have internals must be specified. At this stage data must be specified for the various data sources, code must be given to specify the output functions of the various processes, and compound boxes must be decomposed into their constituent sub-boxes (which may, in themselves, be compound boxes). The contents of a data source will be the data which is to be used to test the model. Specifying this data may involve making assumptions about perceptual processes and the representation of data which is delivered to the model. With regard to processes, these may, in the first instance, operate as look-up tables. Given that the test data is specified, processes may initially be specialized so as only to respond appropriately to this data. Once the complete model has been debugged and is operational, the output functions of processes can be generalized (keeping in mind that their input/output characteristics in the original domain must be preserved). Compound boxes must be specified recursively. That is, for each compound box the three stages outlined here must be repeated, until all compound boxes at all levels have been decomposed into networks of primitive boxes.

Compound boxes are effectively a way of bracketing a part of the model as a complete sub-model. Given this, a second appropriate development methodology involves developing detailed specifications of compound boxes in isolation. As noted above, the interaction of a particular compound box with the remainder of the model can be simulated with data sources and data sinks, and in this way a detailed model of one component can be formulated (and executed) before the complete model has been specified.

OOS lessens the problems of confusing theoretically motivated aspects of a program with implementation detail by lessening the distance between the theoretical statement of the theory and its implementation. In doing so, OOS forces the theorist to consider questions which might otherwise have been ignored (such as the access properties of a buffer, or indeed the specification of any properties specific to a particular box). It might be argued that such questions are truly implementation details, and should not concern the psychological theory. This position is justified only if the behavior of the complete model is independent of the precise value of the property in question. Within OOS it is possible to experimentally test such claims. In particular, the theorist can examine the effect of various objects' parameters on the model's behavior. Hence, by systematically varying the relevant parameters and conducting the appropriate simulations the truth of such claims concerning implementation detail can be ascertained. OOS is thus more than simply an implementation tool. By bring implementation claims to the forefront, and by allowing those claims to be tested, OOS can actually inform psychological theorizing.

An Example: A Simple Production System

In order to illustrate the power and simplicity of OOS this section sketches the implementation of a simple production system within the language. Figure 1 depicts a box/arrow diagram corresponding to such a production system. In this figure, hexagons represent processes, rounded rectangles represent buffers, and diamonds represent data boxes. Arrows with standard arrow heads indicate message sending. Arrows with black triangular tails indicate buffer reading. Thus, the process "Resolve Conflicts" reads "Match Memory" and "Refractory Memory" and sends messages to "Refractory Memory" and "Fire Productions".

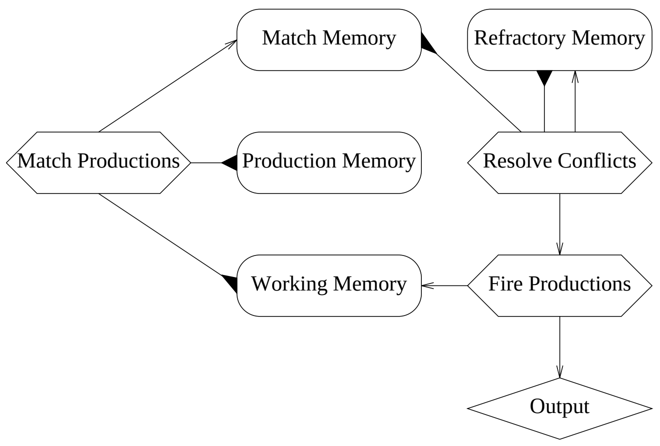

Figure 1: Box/arrow representation of a simple production system.

The diagram is considerably more complex than some production system diagrams, which typically only show working memory and production memory. This is because Figure 1 is complete. It shows *all* processes and buffers, and all communication channels, necessary for a simple production system.

It can be seen then that a production system involves four buffers (i.e., memory components). As well as working memory and production memory, a match memory (in which current instantiations of productions are held) and a refractory memory (in which previously fired instantiations of productions are held) are required. These are all modeled as unlimited capacity decay-free buffers, although OOS allows for the possibility of exploring capacity restrictions or decay characteristics (see Cooper *et al.*, 1993, for details of experiments with working memory and match memory decay in the Soar production system). Each buffer is specified completely in terms of its properties and the subclass of buffer of which it is an instance.

There are three distinct processes. "Resolve Conflicts" is an autonomous process which monitors match memory and refractory memory. When it discovers new production instantiations in match memory which are not in refractory memory, it sends the elements on the right hand side of those instantiations to "Fire Production" and adds the instantiation to refractory memory. "Fire Productions is a triggered process. When it receives a message it adds an element to working memory or sends a message to the output (depending on the message received). These processes are distinct from the "Match Productions" process which monitors working memory and production memory, looking for production instantiations which should be added to match memory. Note that each process is specified locally — as an encapsulated object that responds in a specified way to specified inputs.

The figure embodies a theory of the functional structure of a production system, and there is a direct mapping from the boxes shown to an OOS specification of the system. To transform this specification into a complete implementation it is necessary to specify the properties and subclasses of the various boxes, together with the output functions associated with the various processes. These output functions have each been specified in about a dozen lines of code.

Of course, to model a particular task within the production system it is still necessary to provide task-specific productions. In order to validate the production system described here, it has been tested with the productions necessary for multi-column addition as described by Anderson (1993, p. 10).

Discussion

OOS currently exists as a Sceptic program implementing the execution model and a set of Sceptic libraries implementing the class hierarchy. The libraries have been sufficient for our modeling to date, but elaborations to increase coverage are likely as the language is applied to further domains. For example, at present buffers cannot send messages, but it may be appropriate to include, as a subclass of limited capacity buffers, a class of buffer which sends messages consisting of those elements shunted out when the buffer's capacity is exceeded. Work is also continuing on attempting to further subclassify processes (into classes such as delay, filter, monitor, and agglomerate). This is particularly important as the specification of a process' output function is currently the only substantive programming required in using the language. Lastly, network objects (e.g., feed-forward networks, associative networks, recurrent networks, and interactive activation networks) may also be included, thus allowing the language to be used for modular connectionist and hybrid symbolic/connectionist modeling.

Attempts have previously been made to develop computational tools and modeling environments to assist cognitive modeling (e.g., OPS: Forgy, 1981). Such environments have had little penetration into mainstream cognitive science. The language described here attempts to address the perceived failings of such earlier tools in a number of ways, based on recent advances in computer science. Firstly, by taking functional modularity as the major design requirement, OOS is closer to the formalisms used by traditional cognitive psychologists than, for example, languages based on production system. Secondly, OOS aims to minimize the programming (and hence computational sophistication) required of its uses. Early modeling environments were generally most successful with those skilled in computer languages. OOS still requires a certain level of computational expertise (in specifying the output functions of processes), but this is strictly limited, and as noted above it is anticipated that the inclusion of more object classes will further reduce the programming skills required. Furthermore, the language is well suited to a graphical interface, and a preliminary version of a Graphical OOS Editor (GOOSE) which allows box/arrow diagrams to be drawn and automatically converted from the diagrammatic form into OOS syntax has been developed. This further simplifies the modeling process, taking much of the burden of writing textual code off the psychologist. Work is continuing on a more sophisticated version of this interface.

The use of an object-oriented language for cognitive modeling raises a number of issues. Firstly, it should be noted that the contribution of object-orientedness is more than implementational. Although object-orientedness was motivated on the grounds of providing an appropriate implementation base, the use of a class hierarchy could (if sufficiently extensive) provide a standard specification of box types. Such a specification would lessen the problem of ambiguity and underspecification faced by current uses of box/arrow diagrams, and thereby increase communicability of theories. Furthermore, by associating properties with box classes, issues concerning the relation between theory and implementation (and specifically if certain properties are theoretically relevant) can be addressed.

There is also a question of whether more standard object-oriented languages (such as C++ or Smalltalk) would be more appropriate than OOS for cognitive modeling. In one sense, the language in which the class hierarchy is implemented is irrelevant, but implementing it in Sceptic does bring the benefits of certain primitives common in cognitive theories. The execution model could similarly be implemented in any language — it is, after all, effectively just a shell. It is the properties of that shell that are important, and it is this, which identifies OOS as an object-oriented language specialized for cognitive modeling.

Programmers often abuse languages by using them in ways which conflict with their design aims, and the use of an object-oriented language cannot enforce the development of object-oriented programs. It is, however, difficult to misuse OOS. The underlying execution model is built around the notion of communicating objects. To avoid the use of such objects requires substantial knowledge about the underlying implementation of the execution model. This issue is further addressed by GOOSE, which places tight restrictions on text based programming. The complete model (apart from the output functions of processes) must be expressed in box/arrow notation. The only possible abuse within GOOSE is to overload processes (i.e., to have one process performing functionally distinct operations). The converse of this is that the language may be too restrictive, not allowing sufficient freedom to implement certain classes of models. In general, this difficulty can be addressed by extending the class hierarchy as necessary. Such extensions do, of course, require substantial computational sophistication, and could not be achieved unaided by OOS' target audience.

Lastly, it might be objected that use of OOS implies a commitment to boxes in the brain with messages (and an attendant language of thought) being sent between them. This is not the case. Implicit in the concept of functional modularity is the differentiation of structure and function. Functional modularity does not imply structural modularity: a functional sub-system is a system at the cognitive level and it need not correspond to any identifiable structural sub-system (either neurally localized or neurally distinct) at the neurophysiological level.

Conclusion

OOS, an object-oriented language for computational modeling, has been described. The language facilitates the modeling process by providing a set of object classes appropriate for (symbolic) cognitive modeling within the box/arrow tradition. The language is primarily intended to simplify the development of computational models and thereby empower computationally less sophisticated researchers. The language is also appropriate for the fast prototyping of models and theory driven experimentation (by varying properties or classes of boxes within a model).

Acknowledgements

I am grateful to John Fox, Sofka Barreau, John Morton, Nick Braisby, Tim Shallice, David Glasspool, Bradley Franks, Jonathan Farringdon and two anonymous referees for discussion of, and/or comments on, this work. This work was supported by the Joint Council Initiative in Cognitive Science and Human-Computer Interaction, project grant #G9212530. Further details of this project may be obtained from our Web page at `http://www.psychol.ucl.ac.uk/research/adrem/adrem.html`.

References

Anderson, J. R. (1993). *Rules of the Mind*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Barnard, P. J. (1985). Interacting cognitive subsystems: A psycholinguistic approach to short-term memory. In Ellis, A. (Ed.), *Progress in the Psychology of Language*, (ch. 6, pp. 197–258). Hillsdale, NJ: Lawrence Erlbaum Associates.

Burgess, N. & Hitch, G. J. (1992). Toward a network model of the articulatory loop. *Journal of Memory and Language*, *31*, 429–460.

Cooper, R. & Farringdon, J. (1993). Sceptic Version 4 User Manual Tech. Rep. UCL-PSY-ADREM-TR6, Department of Psychology, University College London, UK.

Cooper, R., Fox, J., Farringdon, J. & Shallice, T. (1993). Towards a systematic methodology for cognitive modeling. Tech. Rep. UCL-PSY-ADREM-8, Department of Psychology, University College London, UK. To appear (subject to revision) in *Artificial Intelligence*.

Fodor, J. D. & Frazier, L. (1980). Is the human sentence parsing mechanism an ATN? *Cognition*, *8*(4), 417–459.

Forgy, C. L. (1981). OPS5 User's Manual. Tech. Rep. CMU-CS-81-135, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania.

Frazier, L. & Fodor, J. D. (1978). The sausage machine: A new two-stage parsing model. *Cognition*, *6*(4), 291–325.

Hajnal, S., Fox, J. & Krause, P. (1989). Sceptic User Manual: Version 3.0. Tech. Rep., Advanced Computation Laboratory, Imperial Cancer Research Fund, London, UK.

Lichtheim, L. (1885). On aphasia. *Brain*, *7*, 433–484.

Miikkulainen, R. (1993). *Subsymbolic Natural Language Processing*. Cambridge, MA: MIT Press.

Morton, J. (1981). The status of information processing models of language. *Philosophical Transactions of the Royal Society of London B*, *295*, 387–396.

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

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. & Lorensen, W. (1991). *Object-Oriented Modeling and Design*. Englewood Cliffs, NJ: Prentice-Hall.

Shallice, T. (1988). *From Neuropsychology to Mental Structure*. Cambridge, UK: Cambridge University Press.

Wermter, S. & Lehnert, W. G. (1989). A hybrid symbolic/connectionist model for noun phrase understanding. *Connection Science*, *1*(3), 225–272.