Formal Verification of Cognitive Models

A. Macklem, F. Mili
Oakland University
Rochester, MI 48309-4478
macklem@oakland.edu, mili@oakland.edu

Abstract

Cognitive modeling has outgrown the toy problems of the research labs and is increasingly tackling Industrial size applications. This growth is not matched in terms of software tools and methodologies. In this paper, we propose a systematic automated approach for verifying the correctness of cognitive models. We present a modular specification language, define model correctness, and present algorithms for automatically checking that a model meets its specifications.

1 Introduction

Cognitive models (CM) are slowly but surely leaving the research laboratories and increasingly used in commercial applications such as evaluating user interfaces such as cell phone menus (Amant *et al.* 2004), war gaming for military training (Doesburg, *et al.* 2005, Ritter *et al.* 2003), and virtual actors and elaborate and credible computer game characters (Funge 2000). The size and level of complexity of these models are not matched by the state of the art in the software engineering of cognitive models, which is still in its infancy stage (Ritter et al. 2003); models are still created from scratch with little reuse taking place; the validation of models is exclusively done through extensive testing (Ritter et al. 2000).

The goal of this research is to contribute to addressing the need for rigorous tools and methodologies for the efficient development of reliable cognitive models by developing appropriate specification languages and methodologies (Mili and Macklem 2005), verification techniques to ascertain the validity of a model (Macklem 2005), and methodologies for reusing cognitive models through composition (Gilmer et al 2004). In this paper, we focus on the verification of cognitive models. We start by briefly describing the specification language developed in section 2. We define the notion of correctness of a cognitive model with respect to its specification in section 3, and then describe the verification algorithms in section 4. In section 5, we discuss the implementation of these algorithms. We summarize our results and discuss future extensions of this work in section 6.

2 Specification of Cognitive Models

2.1 Requirements for CM Specifications

In order to verify that a model meets its requirements, these requirements must be captured in some precise, non-ambiguous form, i.e. in some formal specification language. Yet, in the domain of cognitive modeling in particular, and in artificial intelligence in general, there has not been an established tradition or practice to fully capture requirements. We identify three key impediments to the use of formal specifications with CMs, and state how we have addressed each with our patterns and conflicts specification language and methodology.

The tasks modeled are often ill-structured and hard to specify in a closed-form manner. We address this issue by opting for a modal specification language and approach. Tasks are specified in an incremental manner, capturing one aspect at a time. The overall specification is the sum of the partial specifications.

CMs simulate the processes of human behavior and problem solving, they are best qualified as reactive systems whose specifications cannot be adequately captured by state-based specification languages whose focus is on pre-conditions and post-conditions (Bellini, Mattolini, and Nesi 2000, Wang 2003). We address this by adapting a specification language that captures reactive behavior, yet, is intuitive and easy to read and write (as compared to temporal logic, for example).

CMs are bound by two types of requirements: the competency requirement that they must perform the task at hand (e.g. drive, solve a puzzle), and the cognitive requirement that they perform it in a "human-like" manner. The language developed has the expressive power to capture both; the methodology allows us to capture them separately and to verify them independently.

We introduce the specification language briefly and informally using a small example. We consider a cognitive model developed to solve the Tower of Hanoi problem with

three pegs A, B, and C and three disks small, medium and large. The competency requirement for this problem dictates that, upon completion, the disks must all be on peg B in the correct order, i.e.:

"Final configuration= [small, medium, large] on B".

The competency requirement dictates also that all disk movements must be compliant with the rules of the game, and will include constraints such as:

"No disk can be picked up before the last one picked up is put down."

The cognitive requirements, on the other hand, cover conditions that describe the human-like aspects that we need to impose on the model such as:

"there must be some trial and error."

In other words, we are not interested in producing the recursive systematic solution to the problem.

2.2 Specification Language

We view the specification of a CM as a set of constraints on its traces, where a trace is the sequence of observable events perceived by and generated by the CM in the course of executing a task. We adapt the concept of *patterns and conflicts* introduced by Nodine *et al.* (Nodine *et al* 1995) used to specify correct transaction schedules in cooperative settings. Patterns are constraints that specify interleaving between events that *must* take place such that "every pick-up disk d must be followed by a put-down disk d." Conflicts are constraints that specify interleaving that *must never* take place such as "(must never) place disk d_1 over disk d_2 where $d_1 > d_2$." Selecting a language to capture the patterns and conflicts is a tradeoff between expressive power and ease of manipulation –notably verification in our case. In this paper, we use the restricted version of this language by limiting ourselves to regular languages. In other words, patterns and conflicts are captured using regular grammars or finite state automata for example.

We illustrate below the finite automata capturing some of the Tower of Hanoi requirements.

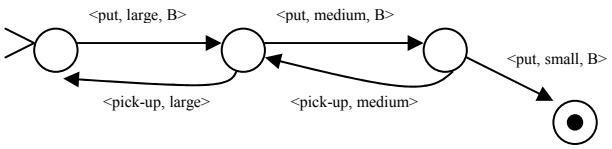

Figure 2.1 Pattern P1: Eventually large, medium, and small disks are place on B

Pattern P1, Figure 2.1, states that eventually the three disks large, then medium, then small are placed on B. The last step: placing the small disk (successfully) terminates the pattern. Note that the initial state in P1 is not an accepting state, capturing the fact that P1 is not satisfied until the three disks make it to their final destination. By contrast, the initial state of pattern P2, Figure 2.2, is an accepting state; but should a disk d be picked up, then the pattern is not satisfied until that disk is placed back. P2 does not require that the disk be put back immediately; thus the need for C1. C1, Figure 2.3, states that once a disk d is picked up, picking up any disk d' (other than d) terminates the conflict (i.e. leads to a conflict). Patterns and conflicts are expressed using the same syntax, but interpreted differently. In a valid trace, every activated pattern terminates, and no conflict terminates.

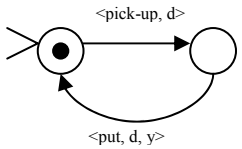

Figure 2.2 Pattern P2: Every pick up disk must be followed by a put disk.

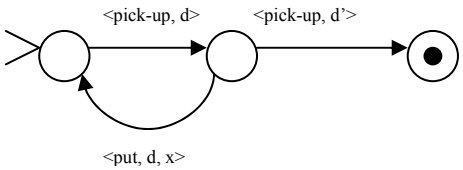

Figure 2.3 Conflict C1: After a disk d is picked up, no other pick up is allowed until d is put down.

A formal definition of the pattern and conflict specification language is available in (Macklem 2005). This example suffices to illustrate an important benefit of this language, namely its ease of use, for both writing specifications and for reading them.

3 Defining Model Correctness

Given a model M, we define the semantics of M to be the set of all possible traces generated by M. A model trace consists of the set of events, actions, and decisions that are either perceived or preformed by the model. Note that this definition of semantics reflects the fact that a cognitive model is characterized as much by its reactive ongoing behaviors as by the final outcomes of its decisions and actions.

The trace of a model is a string composed by symbols of the trace alphabet. The trace alphabet consists of all events, actions and decisions. Typically this alphabet is smaller grain then the specifications' alphabet. For example, for the Tower of Hanoi, the specification alphabet contains symbols capturing "put disk d on peg x", whereas the model trace alphabet consists of symbols capturing decisions such as "select next disk to more". This difference in the level of abstraction is easily addressed by mapping the patterns and conflicts to the model's trace alphabet.

A model is correct with a specification expressed by a set of patterns and a set of conflicts iff:

- Every trace T of the model is correct with every pattern P in the specifications
- Every trace T of the model is correct with every conflict C in the specifications

Intuitively, a trace is correct with a pattern iff the pattern either does not occur or occurs completely. Similarly, a trace is correct with a conflict if the conflict does not fully occur within the trace. A model is correct is *all* of its traces are correct, not just the ones we happened to generate (as in testing).

4 Verification of Cognitive Models

The modularity of the specifications translates into modularity in their verification. In other words, the competency and cognitive requirements can be verified independently. Furthermore, within each of them, each pattern and each conflict can be verified independently.

By definition of correctness, each pattern and each conflict must be verified with respect to *every* trace of the model. Obviously, we will not attempt to generate all the traces individually; instead we build an abstract representation for this set in the form of a rooted graph. The set of traces is the set of all paths from the root of the graph to a leaf node.

4.1 Production Flow Graph

We call the graph representing all traces the Production Flow Graph (PFG), by analogy to the control flow graphs used in program analysis. A PFG is a labeled directed graph G=<V,E,I> characterized as follows:

1. The set of vertices V represents the production rules; there is one vertex r_i for each production rule.
2. For every pair (r_i, r_j), if it is possible for production rule r_j to execute immediately following production rule r_i, then (r_i, r_j)ϵE.
3. For every edge (r_i, r_j) in E, the label l associated with the edge is the action of r_i ■

Note that the above is a *characterization* rather than a strict definition of a PFG in the sense that we are defining a lower bound but not an upper bound for E. We state which edges must be included in E, but allow E to include more than those pairs. This choice allows us to construct PFG graphs easily—and iteratively as illustrated in section 5.

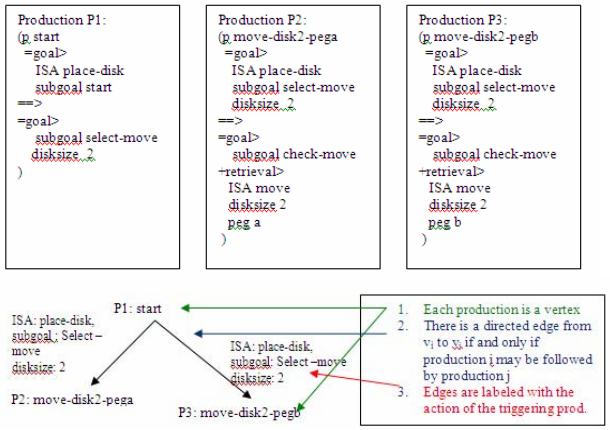

Figure 3.1 Production Rules and Related PFG

Figure 3.1 provides an example of three productions and their corresponding production flow graph. First let's consider the productions in this example. The productions are written using the syntax of the cognitive modeling language ACT-R. The conditions of the production appear before the ==> sign, and the actions appear after it. These productions refer to an important data structure in ACT-R, namely the goal buffer. The goal buffer can have multiple attributes, e.g. ISA, subgoal, disksize. For example, P1's condition is that the current goal has values place-disk and start respectively with no restriction on the value of disksize. Most productions modify the goal buffer in their action clause by modifying the current goal or adding new sub-goals. Production P1 updates the goal by changing the subgoal from start to select-move.

Now, let's consider the related PFG also shown in Figure 3.1. Within the PFG productions P1, P2, and P3 are represented as vertices. A directed edge exists between productions P1 and P2 and between P1 and P3 to illustrate that following P1 either P2 or P3 may be executed. This was determined because the actions of P1 do not conflict with the conditions of P2 and P3. The labels on these two edges encode the action of P1. In fact, as shown in section 5.1, a PFG graph can be generated automatically from the set of productions of a cognitive model.

4.2 Correctness with Patterns

Recall that for a trace to be correct with a pattern, the pattern must not occur or must occur completely within the trace. To show collectively that the set of all possible traces is correct with respect to a pattern, we show that their representation as a PFG is correct with a pattern.

Consider the example shown in Figure 3.2. We can evaluate the correctness of the PFG graph with respect to pattern P1 by determining if each occurrence of P1's prefix (first operation) within PFG is a complete occurrence of the remaining pattern string. If the pattern prefix does not appear within the PFG, correctness is automatic. In the example in Figure 3.2, the prefix of the pattern is the character "*a*". The prefix "a" has two occurrences on the FPG graph, one on edge <1,2> and one on edge <1,8>. For the FPG to be correct with respect to pattern P1 every path rooted at 2 and every path rooted at 8 must complete the occurrence. This condition is satisfied by the paths rooted at 2 (namely: <2,3,4,5> and <2,6,7,5>), as they each contain the remaining pattern string "*b,d*". The same cannot be said about the paths rooted at 8. The only path rooted at 8 is <8,9> and it does not complete the pattern P1. Therefore, the production flow graph PFG is not correct with the pattern specification P1.

The process used for evaluating a PFG's correctness with a pattern, as illustrated with the previous example, was automated. The algorithm starts by searching the PFG graph in a depth-first manner checking for any label that matches the prefix (first operation) of the pattern. If a match is not found, the PFG is correct. If the prefix of the pattern is

found as the label of an edge e_{ij}, then every path leaving vertex j must contain the remainder of the pattern.
One final point to address is the handling of cycles both within patterns and within a PFG. In (Mili and Macklem 2005) we show how we eliminate cycles from patterns. Cycles in the PFG are detected and handled during traversal by the algorithm.

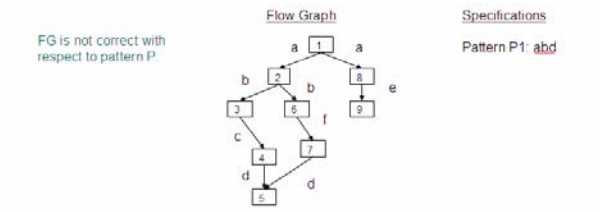

Figure 3.2 Evaluating correctness of PFG with respect to P1

4.4 Correctness with Conflicts

For a trace to be correct with respect to a conflict, the conflict must not fully occur within the trace. As with the patterns, to show collectively that the set of all possible traces is correct with respect to a conflict, we show that their representation as a PFG is correct with the conflict.

Before exploring the comparison process of a PFG with a conflict, we must first address the use of conflict exception strings. A conflict exception string may accompany a conflict string as needed to capture the notion of rolling back an occurrence of a segment of a conflict string. For example,

"*no pick-up small disk can be followed by a put of medium disk unless the small disk is put down first*"

The conflict string would capture the *no pick-up of small disk followed by medium disk* and the conflict exception string would capture the *unless the small disk is put down first*.

Accounting for the possible use of a conflict exception string, means that a PFG is correct with a conflict only if any full occurrence of the conflict within the PFG also contains the conflict exception string.

Consider the example shown in Figure 3.3. We can evaluate the correctness of the PFG with conflict C1 by determining that no occurrence of C1's prefix within PFG is an entire occurrence of the conflict string (this example does not involve a conflict exception string). Prefix *e* occurs first between nodes 8,9 and there are not any paths leaving node 9. Therefore the remainder of the conflict string does not occur. The production flow graph PFG is consistent with conflict specification C1.
The process used for checking a PFG's correctness with a conflict has become the basis for the conflict verification algorithm. This algorithm searches a PFG in a depth-first manner for any edge's label matching the prefix operation of the conflict. If no such label is found, then correctness is automatic. If such a label is found on an edge, the algorithm checks all paths including that edge for the remainder of the conflict string. If it is not found, the PFG is correct with that conflict. However, if any path P contains the entire conflict string then the PFG is not correct with that conflict, unless path P also includes the conflict exception string.

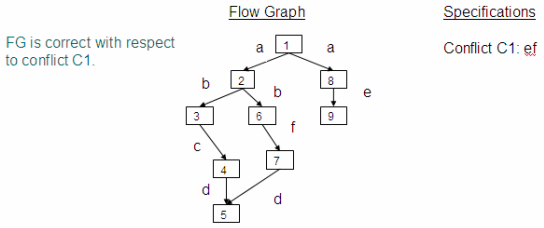

Figure 3.3 Correctness of PFG with respect to C1.

4.5 Verifying PFGs vs. verifying a CM

The focus of this research is to check the correctness of a cognitive model. We have established that it suffices to verify that every trace of the model is correct with respect to every pattern and every conflict. In practice, we do not necessarily have access to the set of all traces, instead, we work with a PFG graph which represents a superset of the traces of the model. The PFG may contain paths that represent traces which would never occur when running the cognitive model. This plays an important role when interpreting the meaning of the correctness between the PFG and the patterns and conflict specifications.

If a PFG is correct with a pattern or conflict then the cognitive model it represents is automatically correct with respect to that specification. This is justified because the PFG contains a superset of the CM traces. On the other hand, a PFG that is not correct with a pattern or conflict does not conclusively indicate that its corresponding CM is not correct as well. Although false positives will never be returned when comparing the PFG and specifications, false negatives may be. It is important for the verifier to use additional semantic knowledge to determine if the path that is non-compliant with the specification may actually be traversed and draw conclusions accordingly.

Aware of the possibilities for false negatives, one can interpret the results of verifying the PFG against each pattern and conflict to deduce the correctness of the CM with respect to those same specifications.

5 Implementation

A key feature of our approach is that the complete process of verifying that a cognitive model meets its requirements can be automated, from the creation of the PFG through the checking of each of the patterns and conflicts.

5.1 PFG Construction

A production flow graph can automatically be created from a cognitive model. To create the production flow graph we use an iterative pruning process. Initially the algorithm constructs a full matrix (full graph) in which all productions have edges connecting to all others. According to the definition of a PFG, a full graph is a valid PFG, although

not very useful! Thus, motivated to create a smaller yet still correct PFG, the algorithm begins an iterative pruning process. It examines one attribute at a time, each time using that attribute to prune edges from the graph. At the i^{th} iteration, we prune every edge e_{jk} if we can determine, based on attribute a_i, that r_k cannot follow r_j in any model trace.

An example of a matrix (representing the production flow graph) is shown before pruning in Figure 5.1 and after three iterations of pruning in Figure 5.2. The initial matrix as shown in Figure 5.1, places an X in each cell to represent an edge between the intersecting row production and column production. As edges are pruned away the cell is cleared. If an edge remains during pruning, the X is replaced with the value of the buffer attribute. When finished pruning, the cell value will be equivalent to the edge label in the graph.

Figure 5.1 Production Matrix – Before Pruning

		Conditions									
Actions		P1	P2	P3	P4	P5	…	P58	P59	P60	P61
	P1	x	x	x	x	x		x	x	x	x
	P2	x	x	x	x	x		x	x	x	x
	P3	x	x	x	x	x		x	x	x	x
	P4	x	x	x	x	x		x	x	x	x
	P5	x	x	x	x	x		x	x	x	x
	…										
	P58	x	x	x	x	x		x	x	x	x
	P59	x	x	x	x	x		x	x	x	x
	P60	x	x	x	x	x		x	x	x	x
	P61	x	x	x	x	x		x	x	x	x

		Conditions									
Actions		P1	P2	P3	P4	P5		P58	P59	P60	P61
	P1		Setup, Peg, Peg A								
	P2			Setup, Peg, Peg B							
	P3				Setup, Peg, Peg C						
	P4					Setup, Disk, Disk 1					
	P5										
	…										
	P58										Move, X, Pick Move
	P59									Move, Disk, Disk 2 Blocked	
	P60										Move, X, Pick Move
	P61										

Figure 5.2 Production Matrix – Pruning with Goal ISA, Retrieval ISA, and Goal Step attributes.

Pruning continue until the verifier is satisfied with the size of the production flow graph. A verifier could choose to exhaust all attributes in order to achieve a syntactically minimal production flow graph. Of course even at this point, semantic knowledge could reduce the graph further.

The justification for iterative pruning of edges is straightforward. To begin all productions have edges connecting to all others. Through the iterations, edges are pruned only when an attribute production's action sets an attribute to a value that does not match the attribute's value given in another production's condition.

The number of attributes used in pruning, as well as the selection of the right attributes to use, will have a reducing effect on the branching factor of the PFG. For instance, some attributes such as the goal or sub-goal are used often in both rule conditions and actions. Therefore, their values are relevant to the majority of production rules and are more likely to facilitate pruning.

For our running example, with 61 productions, the PFG initially has a branching factor of 61. After one pruning iteration the branching factor is on average 24 and after three iterations the largest branching factor is 18 and on average less than 3. This algorithm was also applied to three ACT-R tutorial CMs: unit 4 paired, unit 5 grouped, and unit 6 bst-learn (available at the ACT-R website). Their initial branching factors are 6, 7, and 26 respectively. All three models had an average branching factor of 2 after pruning (with no more than three pruning iterations necessary). Thus, this algorithm is a powerful tool for creating and iteratively reducing the size of the PFG.

5.2 Pattern Verification

The pattern verification algorithm as discussed in section 4.2 is implemented in SQL as a series of three tables and three stored procedures. The time complexity of the pattern verification algorithm is $O(n^d)$, where n is the number of productions and d is the branching factor of the production flow graph. Since the time complexity of this algorithm is exponential with respect to the branching factor, a small branching factor is essential. This is easily ensured by allowing the production flow graph creation algorithm to iterate through as many attributes as necessary to prune the graph into an acceptable size.

5.3 Conflict Verification

The conflict verification algorithm as discussed in section 4.3 has been written in pseudo-code but not yet implemented.

6 Extensions and Conclusions

As cognitive models are used increasingly in industrial settings, we must make applicable the same tools and methodologies applied traditionally in software engineering: specification, validation, reuse, and verification. In this paper, we proposed a verification technique for specifications capturing cognitive model requirements.

We introduced the patterns and conflicts specification language to facilitate the formal specification of CMs in an incremental manner. Patterns capture mandatory interleaving of events and conflicts capture forbidden interleaving of events. We propose that specifications be split into competency and cognitive requirements to enable partial specifications for ill-structured tasks, promote separation of concerns, and allow reuse of specifications across cognitive architectures.

Related works of interest within formal specification and verification include the use of temporal logic (Clarke *et al.* 2000) or process algebra (Wang 2003) to describe human behaviors. Temporal Logic and model checking may be an

approach that can be adapted to the needs of CMs and is an area of research we will pursue. Process Algebra however is heavy in mathematical notations and more complex in nature requiring some mathematical expertise. This highlights one of the most important features of our specification language, the ease in which it can be understood and used to represent requirements. This is an essential feature in order to combat the reluctance to use formal specifications in this "hard to specify" area.

Verification based on formalized specification and automated using simple tools relieves the dependency on the availability of human data or then involvement of a domain expert as necessary with a testing-based approach to validation. Also, this approach provides a means to verify all or partial specifications allowing its use in conjunction with a testing approach.

The limited experience we have had with the proposed approach is encouraging. We are currently working on extensions of this work that include:

- Expanding the expressive power of the patterns and conflicts. For now, we have restricted the syntax to regular expressions which have the dual benefits of being easy to represent and understand graphically and easy to reason about. We are currently exploring the consequences of expanding to context free expressions.
- Although the concepts of patterns and conflicts are intuitive, we have found that first time users misinterpret them an misuse them in some common ways. We are developing specification validation environments in which specifiers are given interactive feedback on their specifications allowing them to refine them and correct them as needed.
- The initial motivation of this work was the possibility of reusing models by composing them together in order to create more complex models. The work presented here is a prerequisite and enabler for such a task.

Additionally, we would like to expand the scope of this research and apply our formal verification technique to other rule-based AI systems.

References

ACT-R website http://act-r.psy.cmu.edu/.

Amant, R. Horton, T., Ritter, F. 2004. Model-based evaluation of cell phone menu interaction, *Proceedings SIGCHI,* 343-350, New York, N.Y. ACM Press.

Bellini, P., Mattolini, R., and Nesi, P. 2000. Temporal Logics for Real Time System Specification *ACM Computing Surveys* (32)1:12-42.

Clarke, E., Grumberg, O., and Doron P. 2000. *Model Checking*. Cambridge, MA: M.I.T. Press.

Doesburg, W., Heuvelink, A., Broek, E. 2005. TACOP: a cognitive agent for a naval training simulation environment. *Proceedings 4th International Conference on Autonomous Agents and Multiagent systems,* 1363-1364. New York, N.Y. ACM Press.

Funge, J. 2000. Cognitive modeling for games and animation. Communications of the ACM, (43)7: 40-48.

Jones, R.M. et al., 1999. Automated Intelligent Pilots for Combat Flight Simulation *AI Magazine* 20(1): 27-42.

Macklem, A. 2005. Verification and Validation of Cognitive Models. Master's Thesis, Dept. of Computer Science and Engineering, Oakland University.

Mili, A., Desharnais, J., and Mili, F. 1994. *Computer Program Construction* Cambridge University Press.

Mili, F., A. Macklem, 2004. Patterns and Conflicts for the Specification, *Proceedings IADIS Conference*, Portugal.

Nodine, M.H., Ramaswarmy, S., and Zdonik, S.B. 1995. A Cooperative Transaction Model for Design Datbases. In *Database Transaction Models for Advanced Applications* Elmagarmid (ed.) Morgan Kaufman:53-85.

Ritter, F. E. et al. 2000. Supporting cognitive models as users *ACM Transactions on Computer-Human Interaction*, 7(2):141-173. http://redeft.ist.psu.edu/papers/ritterBJY00.pdf.

Ritter, F. E., Major, N. P. 1995. Useful Mechanisms for developing simulations for cognitive models. *AI and Simulation of Behaviour Quarterly*, 91(Spring):7-18. http://redeft.ist.psu.edu/papers/ritterM95.pdf.

Ritter, F. E., et al., 2003. *Techniques for Modeling Human and Organizational Behavior in Synthetic Environments: A supplementary Review* Wright-Patterson Air Force Base, OH: Human Systems Information Analysis Center. http://iac.dtic.mil/hsiac/SOARS.htm.

Wang, Y. 2003. Using Process Algebra to Describe Human and Software Behaviors *Brain and Mind* 4(2003):199-213.