

This hierarchical learning of higher-order schemas is an important and a challenging first step in constructivist learning. For instance, Sun and Sessions (2000) proposed the Self Segmentation of Sequence (SSS) algorithm.

In this paper, we show how we could implement such a two-level mechanism in Soar. Our implementation, draws from previous studies that started to implement constructivist learning, especially that of Drescher (1991). Drescher modeled schemas as triples (context, action, expectation). In this paper, for clarity, we sometimes group together action and expectation under the term “intention”. In that description, a schema is a context associated with an intention, and an intention is an action associated with an expectation.

Although these principles are not new, their implementation in Soar is innovative. The Soar community may find useful both our model itself and the remarks it suggests about Soar. More broadly, this work is an illustration of constructivist learning that can be applied to different architectures where an agent has to learn from interaction, such as in teamwork or adversarial interaction.

2. The task

The study of behavior requires a behavior and a task it arises from. This study was started with the simplest possible task. In this task, the agent has a choice between two possible actions: A or B, and he could get two possible responses from the environment: X or Y. This approach may seem like Newell’s (1990) approach when he proposed the “simplest response task” or the “two-choice response task”. However, as early as this stage, our approach is actually different. In Newell’s approach, the task was “reactive”: the subject was previously told what he had to do in response to a stimulus. In our case, the task is “proactive”: the agent does not know what to do but has a built-in “preference” for one of the two responses, namely Y. Our purpose is to study how the agent can learn to do the right action (A or B) to get Y (good) and not X (bad). In other words, our goal is to have the agent structure his behavior due to the fact that he has an innate tendency to prefer actions that will lead to get Y. To achieve this is to have learned a behavior that results in a desired outcome.

We have placed our agent in different environments that implemented this task. The first environment always returned X when the agent was doing A, and Y when the

agent was doing B. So the agent could learn two very basic schemas: Schema1 = (context = any, action = A, expectation = X) and Schema2 = (context = any, action = B, expectation = Y). Because the agent had a hard-coded preference for schemas having an expectation of Y, then after at most two tries, he kept performing Schema2 and getting Y.

Then, we placed our agent in more complex environments where he had to perform more complex sequences of behavior before getting a Y. In these new environments, the connection between action and response was not systematic but depended on the previous context of sequence. This approach can be compared to sequence learning (Sun & Giles, 2000). It is not, however, a passive sequence learning where the agent could only be an observer of the sequence, but it is an active sequence learning where the agent can formulate hypotheses and test them in the environment. This formulation of hypotheses is implemented as a construction mechanism of new schemas. We think that this active approach should facilitate a hierarchical learning of schemas and subschemas. Newly constructed schemas are tested against the environment and reinforced when they succeed. Our approach differs however from a pure “trial-and-error with reinforcement learning” approach by the fact that reinforcement is contextualized into schemas, and schemas are hierarchically organized.

The different environments where we have put our agent are further reported in olivier-georgeon.blogspot.com. In this paper, we only describe an environment where the agent has to do two consecutive A or two consecutive B to get a Y. Continuing with the same action will not lead to Y anymore. So, the agent will only get a Y if he does A when he has previously done B then A, or if he does B when he has previously done A then B. We call this task the “AXAYBXBY” task. This task is designed to illustrate how the agent can learn second order schemas that force him to enact primary schemas with an expectation of X—which is against his hard-coded tendency—to put himself in a situation where he can get a Y at the next round.

3. The learning mechanism

Our implementation of learning can be described as three levels of abstraction made by the agent, above level 0 that is the raw activity. This mechanism is represented in Figure 1.

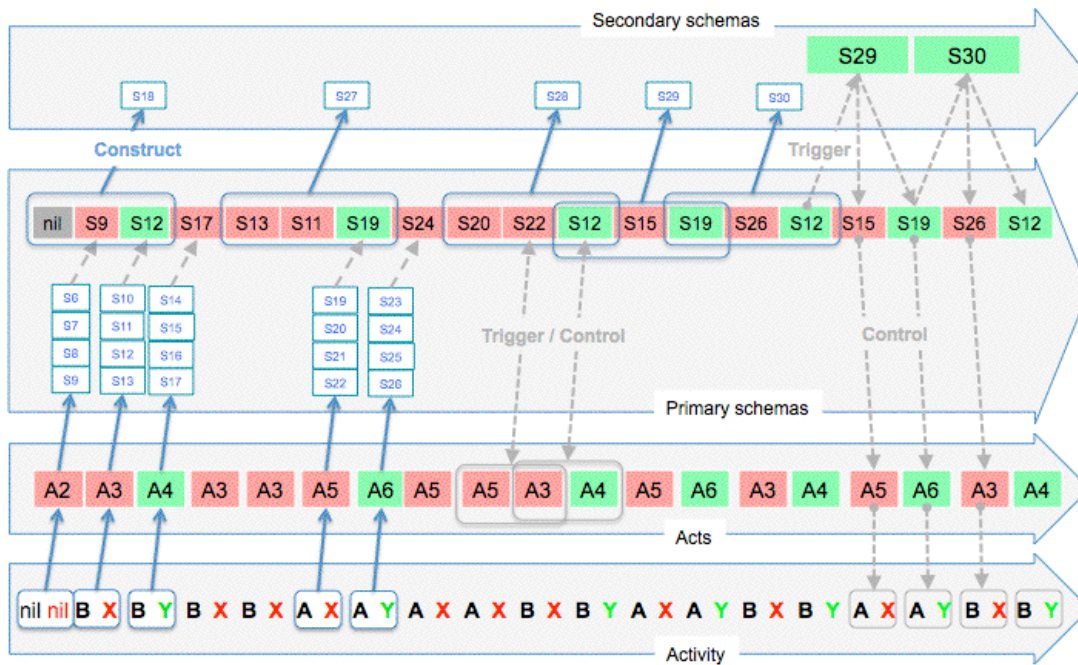


Figure 1: Abstraction process.

The raw activity is a sequence of primary actions (A or B) performed by the agent, followed by primary responses (X or Y) returned by the environment. When we run our agent in the environment, we obtain the raw_sequence = “B X B Y B X B X A X A Y ...”. Note that the first “nil nil” elements at the beginning of Figure 1 represent the content of the agent’s short-term memory at the beginning of the experiment and does not correspond to any real action.

The first abstraction level consists of having the agent group primary actions with their primary responses in what we call an *act*. For example act A3 = “B X”. The first abstraction of raw_sequence is then act_sequence = “A3 A4 A3 A3 A5 A6 ...”

The second abstraction level consists of having the agent group two acts into one *primary schema*. For example, Schema S12 = “A3 A4”. A3 is the *context act* of S12, and A4 is its *intention act*. That means that, in a context where A3 has just been performed, S12 proposes to do A4. A4 consists of doing the primary action “A” and expecting a response of “Y”. Thus, the second abstraction level of raw_sequence is primary_schema_sequence = “S12 S17 S13 S11 S19 ...”

The third abstraction level consists of grouping three primary schemas into one *secondary schema*. For example, Schema S27 = “S13 S11 S19”. S13 is the context of S27. S11 is the action of S27, and S19 is the expectation of S27. S11 and S19 together form the *intention* of S27. That means that, in a context where S13 has just been performed, S27 proposes to enact S11, which should (if it succeeds) lead to the possibility of

enacting S19, which should (if it succeeds) lead to getting a Y.

In Figure 1, the ascendant solid arrows represent the construction of more abstract items by the agent. The relation between higher level and lower level is somewhat complex, globally higher levels tend to “control” lower levels, but sometimes they fail, and lower levels “trigger” higher level schemas, but it is not certain what higher level schema is performed until the environment has responded and the schema is completely over. When the actually performed schema is known, its weight is incremented. This weight is then used during the selection phase for selecting the action that lead to a Y. Dashed gray double-headed arrows “Trigger/Control” represent this tightly coupling between levels. In the “AXAYBXBY” environment, secondary schemas always succeed, so the agent becomes “in control” of his activity when secondary schemas start to be enacted. In Figure 1, that is shown by “S29 S30” in green that control the enaction of S15, S19, S26, S12, which finally control the sequence AXAYBXBY. This sequence is the beginning of a stable sequence that gives a Y to the agent every second round, which is the best he can get in this environment.

4. The Agent

We have modeled this agent and its environment in Soar 9.1, and executed it with the Soar debugger. So the agent and his environment are firing each cycle in turn, thanks to a token mechanism, but they are both part of the same overall Soar system. This approach avoids tying the model to an external task at this point, an important, but

complicated aspect of model development (Ritter, Baxter, Jones, & Young, 2000).

Each round, the agent executes one of the two green clockwise loops in Figure 2, and the environment executes the blue counterclockwise loop. At the beginning, the model starts from the “Construct Context” phase. The context corresponds to the current situation as it is retained in the agent’s short-term memory (“nil” at the beginning).

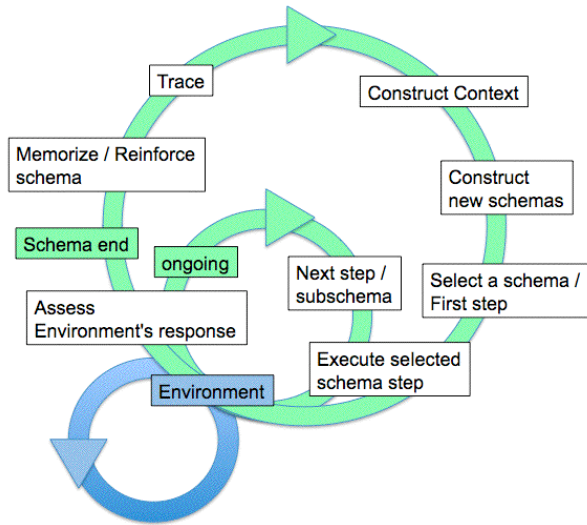


Figure 2: Execution cycle.

Construct context: structures the current context to prepare the schema construction and the schema selection. The context is made up of the three previously enacted schemas, stored in short-term memory. These schemas can be of any level and can refer to subschemas. This phase indexes these different levels.

Construct new schemas: creates new schemas that match the current context. If they do not yet exist, these new schemas are added to long-term memory. They constitute hypotheses about how to deal with a new context, but they still need to be tested.

Select a schema / First step: selects a schema to be executed in this context. High-level schemas add weight to their subschemas. Weights are positive if they lead to Y and negative if they lead to X. Schemas of any level compete, and the one with the highest weight is selected. If there are several equivalent, one of them is randomly picked. This phase initialize the selected schema at its first step.

Execute schema step: sends the selected action defined in the current schema step to the environment: A or B. (Here the token is passed to the environment.)

Environment: computes the response from the environment and sends it (Y or X) to the agent. The environment has its own memory and cycle. (Here the token is passed back to the agent.)

Assess environment's response: checks if the schema has succeeded or failed. If the current subschema has

succeeded but it is not the last step of the selected schema, then the "ongoing" loop is selected.

Next step / subschema: selects the next step in the subschema hierarchy of the selected schema.

Memorize / reinforce schema: when the schema ends, if it has succeeded, then it is referred to as the last enacted schema in short-term memory. The previous two are shifted, and the previous third is drop out of short-term memory. If the schema has failed at some point, then the actually enacted schema is stored in short-term memory and reinforced in long-term memory. For example, if a primary schema expecting Y has been selected, but if the environment actually returned X, then an equivalent schema but with an expectation of X is actually memorized and reinforced. The reinforcement consists of adding 1 to the schema weight.

Trace: is only used to generate the trace of this cycle and to clear the temporary data.

4.1 The implementation in Soar

Figure 3 shows a part of our model’s memory structure as it is implemented in Soar. There are three main branches: the agent’s memory (<agt>), the environment’s memory (<env>), and a memory structure used for the interface between the agent and the environment (<int>). The agent’s primary schema memory and short-term memory are explicitly represented (<scm> and <stm>). For example, schema S12 = (A3 A4) = (B X B Y) is stored as a subgraph (<sch>) of the schema memory node (<scm>). A3 = (B X) is the context act (<sch>.<con>) made up of the primary action B (<con>.<set>) and the primary response X (<con>.<get>). The second B is the primary action proposed by the schema (<sch>.<set>). Y is the schema expectation (<sch>.<get>). In addition, this schema has a weight (<sch>.<wei>) that is an integer value equal to the number of times this schema has been successfully enacted.

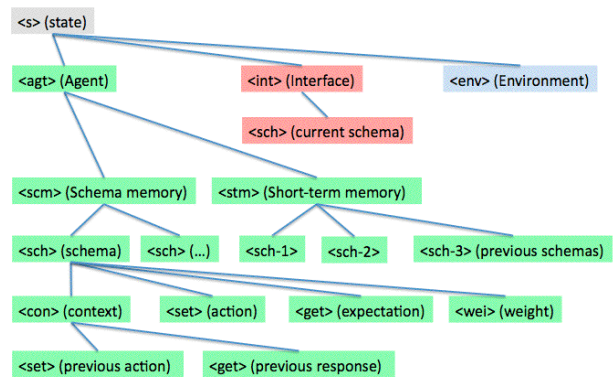


Figure 3: Memory architecture.

Secondary schemas are stored in a different memory structure than primary schemas that is not represented on the figure. Implementing a recursive exploitation of

schemas of any level that would lie on the same memory structure appears difficult at this point in Soar.

5. The behavior study

To help develop this agent and understand its behavior, we have given special attention to his activity traces. We have set up a mechanism to easily format, configure and display them, based on our previous work on understanding activity traces (Georgeon, 2008; Georgeon, Mille, & Bellet, 2006).

This mechanism exports activity traces from the Soar debugger to text files. It converts these text files into XML files using a Java program. These XML traces are filtered to retain only the useful information at a specific step of the study. The XML files are then displayed in a browser using stylesheets. These stylesheets are both XSL and CSS. XSL allows us to specify in which blocks the trace elements will be rendered, and to define a page setup. CSS allows us to specify which format will be used for each block: font, colors, size, margin, etc. This process automatically generates traces like that shown in Figure 4. When displayed with Firefox, these traces can be spoken aloud, which makes them easier to understand. This can be seen online at olivier-georgeon.blogspot.com. The trace of Figure 4 is a subpart of the trace drawn in Figure 1 that focuses on the construction and enaction of secondary schemas.

Context (S11, S19, S24=(A6, A5=(A, X)))
Enact primary S20 = (A5 A X) (1) O-o
Context (S19, S24, S20=(A5, A5=(A, X)))
Enact primary S22 = (A5 B X) (1) O-o
Context (S24, S20, S22=(A5, A3=(B, X)))
Enact primary S12 = (A3 B Y) (2) Yee!
Context (S20, S22, S12=(A3, A4=(B, Y)))
Construct secondary S28 = (S20 S22 S12) (1)
Enact primary S15 = (A4 A X) (1) O-o
Context (S22, S12, S15=(A4, A5=(A, X)))
Enact primary S19 = (A5 A Y) (2) Yee!
Context (S12, S15, S19=(A5, A6=(A, Y)))
Construct secondary S29 = (S12 S15 S19) (1)
Enact primary S26 = (A6 B X) (1) O-o
Context (S15, S19, S26=(A6, A3=(B, X)))
Enact primary S12 = (A3 B Y) (3) Yee!
Context (S19, S26, S12=(A3, A4=(B, Y)))
Construct secondary S30 = (S19 S26 S12) (1)
Enact secondary S29 = (S12 S15 S19) (2) Yee!

Figure 4: XML activity trace.

Each round ends by a red or a dark green line: red when the agent gets X (bad, he says O-o at the end of lines) and dark-green when he gets Y (good, he says Yee!). Light-green lines are intermediary steps of context construction and schema construction. For example the context at the top is made up of the three previously enacted schemas

S11, S19, and S24. S24 is expanded as acts A6 and A5. A5 is expanded as primary action A and primary response X. Below, when S12 succeeds, we can see the construction of a secondary schema S28 made up of the three previously enacted schemas S20, S22, S12. At the bottom of the figure we can see the first enaction of a secondary schema S29, because its context primary schema S12 matched the previously-enacted primary schema.

6. Results and discussion

The interesting results that we think this study has brought are:

- We have defined a simple task suitable to study bottom-up learning. We have named this task the “AXAYBXBY” task.
- We have implemented an agent that can learn two levels of schemas to perform the “AXAYBXBY” task as well as other related tasks (reported in the website), namely, all tasks with a regularity span lower or equal than two rounds.
- We have shown that this agent could be implemented in Soar, which is a cognitive architecture that is better known for supporting learning from higher-level impasses (through chunking) than bottom-up learning from actions.
- We have developed a mechanism to generate and display activity traces from Soar debugger logs. We show that it is useful that the modeler has the possibility to easily configure and format activity traces during the modeling process.
- We have illustrated a constructivist approach of learning, inspired from a conception of cognition where the basic elements are not declarative chunks but schemas that can be seen as “contextualized patterns of behavior”. As said in the introduction, this approach is not novel, but it is the first time it has been implemented with Soar.
- More broadly, we think that this approach can illustrate psychological phenomena related to learning and activity control. For instance, it is interesting to notice that the cycle represented in Figure 2, which has been defined from purely logical and ad-hoc engineering purpose, can be related to more psychological theories, namely the OODA loop (Hammond, 2001).
- Our implementation led us to several remarks on how Soar could be used in this approach. These remarks are listed below.

6.1 Remarks on how Soar is used in this approach

Implementing this in Soar led to a deeper understanding of Soar and how to build models efficiently with Soar.

We do not use Soar's input and output functions. The Soar system implements both our agent and his environment. From the Soar viewpoint, this model does not interact with any outside environment; it evolves by interaction between subcomponents of the system. It is only us, as

observers, who understand it as an agent interacting with an environment. This approach suggests that there may be a sub-type of environment where the environment's behavior and actions can be represented using the same theory of what is in the head. We term such environments 'cognitive' environments. Early work on problem solving used these environments, such as small towers of Hanoi, missionaries and cannibals, and water jugs.

Our agent's memory does not match the classical Soar memory definition. From our agent's viewpoint, he stores schemas in his long-term memory, and the current situation in his short-term memory. From the Soar viewpoint, however, these schemas and this situation are actually stored in what the Soar vocabulary calls working memory, usually considered as declarative and semantic. Thus, Soar modelers could think that our agent learns semantic knowledge, but that would be seen by many as inappropriate because, from our agent's viewpoint, this knowledge has no semantics, it is only behavioral patterns and thus should be seen as a type of procedural knowledge.

We do not describe our agent's action possibilities as operators, contrary to many Soar models. Instead, we describe them as schemas, and our model dynamically creates operators to generate the appropriate action that results from the evaluation of schemas.

We cannot use Soar-9's built-in reward mechanism for two reasons. The first is that it only applies to operators, and we do not need to reinforce operators but schemas. The second is that Soar reinforcement learning is designed to let the modeler define rewards from the environment. From these rewards, Soar computes operator preferences through an algorithm on which we have insufficient control. In our case, our agent's behavior is not driven by rewards sent to him as inputs (and that would be backward propagated through an algorithm like the bucket brigade algorithm), but by hard-coded preferences for schemas having certain types of expectation. Therefore, the Soar reward mechanism does not help us, and we have to implement our own reinforcement mechanism that just increments a schema's weight each time it is enacted. We use, however, the numerical preference mechanism available in Soar-9, to select the operator that receives the highest overall weight from the different schemas that support it.

So far, we do not use the Soar impasse mechanism. In our approach, when the agent has no knowledge to help him choose between two or more schemas, that really means that he has no other thing to do than randomly pick one.

We do not use the Soar's default probabilistic action selection mechanism. The idea that there should be an epsilon probability that our agent chooses not his preferred action is useless in our case. It only impedes the

exploration and learning process. We force the epsilon value to zero. A non-null epsilon value could however be useful in the environment's model, if we would like to test our agent in a noisy environment. But even in a noisy environment, we do not think it is useful to add noise in the agent himself, because, in our approach, the necessary stochastic exploration comes from the random action when there is no preferred schema.

From these remarks, it is clear that our usage of Soar does not correspond completely to what Soar has been designed to support. Soar has been created for representing the modeler's knowledge but does not fully support our approach for developing agents who construct their own knowledge from their activity. Nevertheless, so far, Soar has proven to offer enough flexibility to be usable for this approach. It provides us with powerful and efficient graph manipulation facilities, and weight preference management that are essential for our agent. Further work may align this approach more directly with how Soar's mechanisms are typically used, or may provide more robust suggestions for how they should be used.

6.2 Difficulties

Our mechanism of schema construction may require some more discussion; it is different between primary and secondary schemas. The construction of primary schemas is implemented through a combination of all the possible primary actions that could be done in a specific context with all the primary responses that can be expected. This is illustrated in Figure 1 by the blue upward arrows. For example, in context A3, the agent constructs four schemas S10, S11, S12, S13; respectively of doing A, expecting Y; doing A, expecting X, doing B, expecting Y; and doing B expecting X. In contrast, the construction of secondary schemas is made from the memory of which patterns of primary schema succeeded. For example, secondary schema S27 was constructed because the successful primary schema S19 was enacted. We should explore more generic schema construction mechanism in the future.

In terms of scalability, we should notice that the number of new schemas constructed at each round would not grow with the environment complexity but with the agent's complexity, which remains under the modeler's control. The time needed to explore the environment would however grow with the environment complexity. This raises the interesting question of the agent's "education", that is, designing "pedagogical" situations where the agent could more easily learn lower-level schemas on which higher-level schemas could anchor.

Another difficulty, although related, is that it is not easy to implement a recursive schema/subschema mechanism in Soar. Soar has not been designed for this kind of

recursivity. If we could complete that, it would provide a way for creating further models that use this approach. It should be pretty general, and particularly applicable to other Soar models.

6.3 Future work

We now need to continue implementing higher-level abstraction mechanisms in our agent and test it in more complex environments. The idea of modeling the environment in Soar is nice as long as we don't need a spatial environment. For a 2D environment that provides a screen display and 2D interactions, we plan first to use Vacuum (Cohen, 2005). Vacuum is a simple grid environment that allows an agent, represented as a vacuum cleaner, to move in search for dust to clean up. Then we plan to use dTank (Ritter, Kase, Bhandarkar, Lewis, & Cohen, 2007). dTank is a lightweight environment that simulates a battlefield where two teams of tank can compete. Putting our agent in these environments will allow us to compare it with previous agents that have been developed through classical methods. This will help quantify how less knowledge the modeler has to encode in the agent with our approach.

We also plan to implement a speech mechanism in real time, instead of having the trace spoken when it is analyzed. We believe this would help modelers and users better understand the agent.

7. Conclusion

We have proposed an approach that offers a way to implement unsupervised learning that occurs while a model implemented in Soar is performing a task. This approach uses reinforcement learning to improve performance and to generate a summary of behavior. It does so with declarative representations used to generate procedural behavior, which is relatively novel.

One could think that Soar is not designed to model subsymbolic processing. However, from our agent's viewpoint, this low-level behavior organization can be seen as subsymbolic. In our approach, the learned knowledge is not computed as it is in classic computational approaches. Knowledge of how to perform a task is not binary in that the agent does or does not know how to perform a task at a given point in time. Our approach provides more graded knowledge representations. We do not claim that our agent "has symbols in his mind", nor that he is "mindful" at all. We think however that this first step of bottom-up learning is a required step before modeling agents that are able to manipulate symbols that are grounded in their activity as well as the world, and thus, in a pragmatic conception of knowledge, agent for which these symbols make sense. So far in this study, we demonstrated that a symbolic

cognitive architecture like Soar supports working towards this goal.

This learning mechanism is important in at least three ways. First, it provides a way for procedural knowledge to be learned by an agent through a trial-and-error and the reinforcement mechanism. Second, the abstraction mechanism opens a way for declarative knowledge to be learned by an agent about its procedural knowledge. This is a novel result. Third, in our approach, the environment can be seen as another agent, thus this approach can provide a way for an agent to learn about another agent. This type of knowledge is useful for understanding teammates or opponents.

8. References

- Anderson, J. R., & Lebière, C. (1998). *The Atomic Components of Thought*. Hillsdale, NJ: Lawrence Erlbaum Associates.
- Bach, J. (2003). *The MicroPsi agent architecture*. Paper presented at the ICCM-05, Universitäts-Verlag Bamberg. 15-20.
- Chaput, H. H. (2004). *The Constructivist Learning Architecture: A model of cognitive development for robust autonomous robots*. Unpublished doctoral dissertation, The University of Texas, Austin.
- Cohen, M. A. (2005). Teaching agent programming using custom environments and Jess. *AISB Quarterly*, 120(Spring), 4.
- Cohen, M. A., Ritter, F. E., & Haynes, S. R. (2005). *Herbal: A high-level language and development environment for developing cognitive models in Soar*. Paper presented at the 14th Conference on Behavior Representation in Modeling and Simulation, Orlando, FL. 177-182.
- Drescher, G. L. (1991). *Made-up minds, a constructivist approach to artificial intelligence*. Cambridge, MA: MIT Press.
- Georgeon, O. (2008). Analyzing traces of activity for modeling cognitive schemes of operators. *AISB Quarterly*, 127, 1-2.
- Georgeon, O., Mille, A., & Bellet, T. (2006, 4-8 Sept 2006). *Analyzing behavioral data for refining cognitive models of operator*. Paper presented at the Philosophies and Methodologies for Knowledge Discovery, Seventeenth international Workshop on Database and Expert Systems Applications, Krakow, Poland. 588-592.
- Hammond, G. (2001). *The mind of War: John Boyd and american security*. Washington, DC: Smithsonian Institution Press.
- Harnad, S. (1990). The symbol grounding problem. *Physica D*(42), 335-346.
- Haynes, S. R., Cohen, M. A., & Ritter, F. E. (2009). Design patterns for explaining intelligent systems. *International Journal of Human-Computer Studies*, 67(1), 99-110.

- Laird, J. E., Gongdon, C. B., & Coulter, K. J. (1999). *The Soar User's Manual Version 8.2*. University of Michigan.
- Langley, P., & Choi, D. (2006). Learning recursive control programs from problem solving. *Journal of Machine Learning Research*, 7, 493-518.
- Newell, A. (1990). *Unified Theories of Cognition*. Cambridge, MA: Harvard University Press.
- Piaget, J. (1937). *The construction of reality in the child*. New York: Basic Books.
- Ritter, F. E., Baxter, G. D., Jones, G., & Young, R. (2000). Supporting cognitive models as users. *ACM Transactions on Computer-Human Interaction*, 7(2), 141-173.
- Ritter, F. E., Kase, S., E., Bhandarkar, D., Lewis, B., & Cohen, M. (2007). *dTank updated: Exploring moderator-influenced behavior in a light-weight synthetic environment*. Paper presented at the 16th Conference on Behavior Representation in Modeling and Simulation, U. of Central Florida: Norfolk, VA. 51-60.
- Sun, R. (2004). Desiderata for cognitive architectures. *Philosophical Psychology*, 17(3), 341-373.
- Sun, R., & Giles, C. L. (2000). *Sequence Learning - Paradigms, Algorithms, and Applications* (Vol. 1828). Berlin Heidelberg: Springer.
- Sun, R., Peterson, T., & Merrill, E. (1999). A hybrid architecture for situated learning of reactive sequential decision making. *Applied Intelligence*, 11, 109-127.
- Sun, R., & Sessions, C. (2000). Automatic Segmentation of Sequences through Hierarchical Reinforcement Learning. In R. Sun & C. L. Giles (Eds.), *Sequence Learning* (pp. 241-263). Berlin Heidelberg: Springer-Verlag.
- explanation facilities for intelligent systems, and design rationale.

Acknowledgments

Support was provided by ONR Contracts N00014-08-1-0481 and N00014-06-1-0164.

Author Biographies

OLIVIER GEORGEON is a cognitive scientist with an interest in learning from experience; he is currently a research associate in the ACS (Applied Cognitive Science) Lab in the college of IST at Penn State.

FRANK RITTER is on the faculty of the College of IST, an interdisciplinary academic unit at Penn State to study how people process information using technology. He edits the *Oxford Series on Cognitive Models and Architectures* and is an editorial board member of *Human Factors*, *AISBQ*, and the *Journal of Educational Psychology*.

STEVEN HAYNES is a Professor of Practice in the College of IST at Penn State. His research focuses on