Machine Self-Consciousness More Efficient Than Human Self-Consciousness?

Tristan Cazenave
LIP6, Tour 46-00 2ème étage
Université Pierre et Marie Curie, 4, place Jussieu
75252 Paris Cedex 05 France
e-mail: Tristan.Cazenave@lip6.fr

Abstract

An artificial system that introspects itself and improves itself has written another system that gives better results than systems directly written by people that incrementally creates model of expertise : cognitive scientists, Artificial Intelligence researchers and experts of the domain it has been applied to. It has been applied successfully to the game of Go, to multi-agent simulations and to other domains. This is an encouraging result for researchers working on modeling consciousness, it proves that a machine model of consciousness can be more efficient in creating complex cognitive models than the consciousness of an expert.

1 Introduction

Using self-consciousness, human beings can have some knowledge about their own behaviors ; this knowledge is very useful and self-consciousness is more developed in human beings than in any other animals. According to [Pitrat 1990], this is the reason why we overestimate our potential for being self-conscious. The mechanisms of consciousness could be much more powerful. [McCarthy 1996] gives some mechanisms that conscious machines should have. Introspect [Cazenave 1996b] is a system that experiments some of these mechanisms for real. The system can observe its own behavior so as to detect its own inefficiencies and repair them. When it detects inefficiencies, it reasons on its reasoning process so as to understand why it has been inefficient. Then it modifies itself so as not to be inefficient again in similar situations. Introspect has mainly been used to discover knowledge in the game of Go. It has only been given the rules of the game, by playing the game, it has observed, understood, then modified itself and has discovered a lot of useful expert Go knowledge. It has found by itself more Go knowledge and more useful Go knowledge than some of the best experts of the game of Go associated to computer scientists that have worked during years on the problem.

Consciousness is very difficult to explain because we know we are conscious, but we cannot observe how we are conscious. My opinion is that consciousness is the observation of a system (for example short term memory) by another one. But we are limited in the observations by the low capacity of our short term memory. To understand itself, consciousness would have to observe another system, and using the same mechanism, to observe itself observing the subsystem while dynamically changing. This kind of self self-observation may be of limited interest in everyday life (except for Artificial Intelligence researchers and Cognitive Scientists), and would need a much larger short term memory than we have. That may be the reason why we are unaware of how consciousness works in our minds.

We know that we are conscious, but it is very difficult, because of our limits, to explain what is consciousness, and even to define it. In this paper, we will study consciousness as the ability for a system to observe itself, reason about itself and make appropriate changes of itself so as to improve itself. Some attempts to explain consciousness from a cognitive science point of view [Dennett 1991] are interesting for defining useful concepts and tools to understand consciousness, but they lack of experimental foundations and of scientific facts for proving their assertions. We strongly believe that the best way to understand consciousness is to build a model of it, to make it run on a computer, and to incrementally refine it by comparing its behavior to ours (that is the way good computer Go system are made, but we will explain that later in the paper). This is not the easy way, but this is the best way to prove scientific and reproducible facts about consciousness as a property of a complex system [Sloman 1996].

In section 2, we explain why computer Go is a good domain to study introspection. In section 3, we explain how Go experts and Artificial Intelligence researchers build computer models of Go players. Section 4 describes the Introspect system, an implemented model of introspection. Section 5 is oriented toward the logic language used in our model of introspection. Section 6 is about the usefulness of unconsciousness. Section 7 gives the results of Introspect. Section 8 outlines promising areas for future work.

2 Computer Go and Introspection

This section describes the game of Go. It briefly describes how are made actual Go program and stresses the interest of the game of Go for machine introspection.

Go was developed three to four millennia ago in China; it is the oldest and one of the most popular board game in the world. Like chess, it is a deterministic, perfect information, zero-sum game of strategy between two players. In spite of the simplicity of its rules, playing the game of Go is a very complex task. [Robson 1983] proved that Go generalized to NxN boards is exponential in time and [Allis 1994] shows that Go is the most complex two persons complete information game. It is impossible to make a brute force search of all the moves in the game of Go, the best Go playing systems rely on a knowledge intensive approach. A Go expert uses a large number of rules. Go programmers usually try to enter these rules by hand in a Go program. Creating this large number of rules requires a high level of expertise, a lot of time and a long process of trial and error. Moreover, even the people who are expert in Go and in Computer Science find it difficult to design these rules. This phenomenon can be explained by the high level of specialization of these rules: once the expert has acquired them, they become subconscious and it is hard and painful for the expert to explain why he has chosen to consider a move rather than another one. He is not conscious of all the reasons why he has chosen the right move. The difficulty of encoding Go knowledge is the consequence of a well known difficulty of expert system development: the knowledge engineering bottleneck. Writing a program that is able to observe itself, to detect its own inefficiencies, to create new knowledge so as to uncover them, and to use the new knowledge efficiently is a nice way to avoid this bottleneck by replacing the knowledge extraction process with an automated learning system based on introspection. Machine introspection and learning enable to get rid of the painful expert knowledge acquisition. Thus, computer Go is an ideal domain to test the efficiency of machine introspection when faced with expert human programmers associated to professional Go players (i.e. human introspection).

3 How people use introspection to model Go players

Every Go programmer who is also a good Go player (and they usually are) is faced with an introspective problem. They use a lot of knowledge to play Go, but they cannot tell the knowledge they use. It is even worse, they build models of themselves they think to be right, but that are in fact quite wrong. When they write their first system, they give the system what they think is useful Go knowledge. Then they make the program play and are horrified by its play and its obvious lack of very simple Go knowledge. Those who can psychologically survive such a dramatic experience (it is dramatic from the Go player point of view, but it is even more dramatic from the philosophical and Socratic point of view : being faced to one's inability, or at least high difficulty, of knowing itself), begin to look at the problem with less self-confidence. After a lot of trial and errors and three to four complete change of their models and rewriting of their program, they usually come to work on their program as described in Figure 1, incrementally refining it.

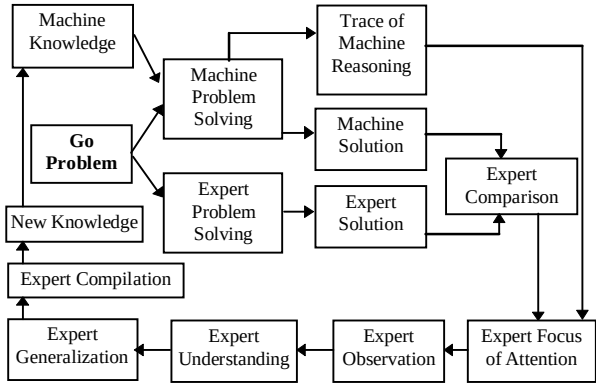

Figure 1

This is what comes out of talks with other authors of good Go programs. This also what comes out from my personal observation of someone creating a computer model of a Go player in my laboratory [Bouzy 1995]. As people are only conscious of their short-term memory, they do not have access to their long-term memory, and therefore do not have a direct access to their knowledge. To access it they have to use it. When the machine tries to solve a problem, it shows the knowledge it has used to solve it. If the machine fails to solve it, and the expert succeed, then the expert can decide to focus its attention on the trace of machine reasoning so as to find the knowledge that the machine does not have and give it. So, by observing the machine reasoning, the expert finds the differences with its own reasoning and understands what is the missing knowledge. When the expert has discovered the knowledge that is missing to solve that particular problem, he tries to generalize it so that it can apply in many more situations. After generalizing the knowledge, it transforms it into a representation that can be efficiently used. Then it adds the new knowledge to the machine knowledge, and removes the knowledge that is no longer useful due to the incorporation of new and more general knowledge.

4 An implemented model of introspection

Figure 2 gives a general view of Introspect. When it is compared to Figure 1, one can notice that the human part of the process has been shifted to the machine. It is now the machine that is in charge to introspect itself so as to improve itself. We will briefly

describe how it is done by succinctly explaining the boxes and arrows of Figure 2.

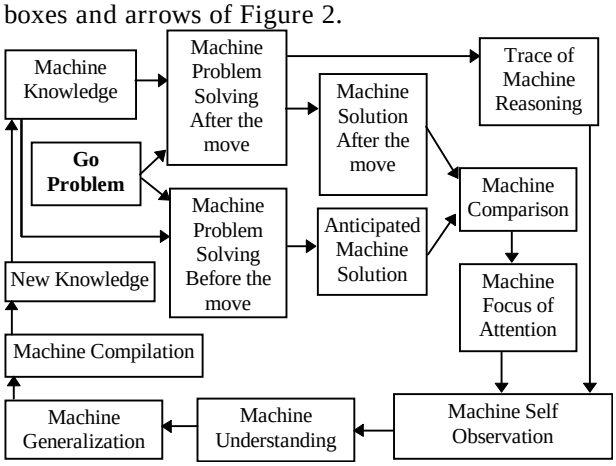

Figure 2

A Go problem is a Go board associated to a move. Go problems comes from books, from recorded games between people and from the games our system has played. When given a Go problem, Introspect does two things. First it deduces everything it can on the board of the problem before the move, and find a lot of anticipated machine solutions to all the problems associated to the board. Then it plays the move and deduces the consequences of the moves, memorizing its deductions into the trace of machine reasoning. It also deduces everything he can on the board after the move. The board after the move is deduced of the board before the move and of the move itself, using the rules of the game of Go represented in first order logic. So it finds a lot of solutions to the problems associated to the board after the move.

The system focuses its attention on the trace of its reasoning when it has a surprise after a move. If it did not succeed in solving a problem before the move was played, and if it can solve it after the move was played. Then the move enables to solve a problem, and it did not see it. So it has to focus its attention on the processes that enabled it to solve the problem so as to explain to itself why the move works.

The system observes its own reasoning if nothing interesting was found before the move, and something happened to be interesting and true after the move. The system can deduce that it has failed to forecast something interesting. It would be good for it to forecast it in similar situations. So it observes its deductions so as to modify itself and be able to forecast it next time in similar situations.

Machine understanding is the explanation the system gives to itself of the reasons why the move works. The system creates the explanation by going backwards in the chain of deductions, replacing facts that describe the board after the move by facts that describe the board before the move (the nonmonotonicity of the domain due to the changes of the board between each move is handled by having an explicit representation of time: each move is associated to a given time). Sometimes, multiple explanations are possible to explain why a fact has been deduced. So the explanation of the interest of the move is a tree. The system cuts the tree of explanations by eliminating the explanations that are subsumed by others at the leaves of the tree. An interesting fact can lead to multiple explanations, and then to multiple rules to add to the system knowledge. Machine generalization is made by replacing some constants in the explanations by variables. The constants to generalize are appropriately chosen by using the rules of the game represented in first order logic. Machine compilation is the automatic ordering of the conditions of the rules so as to minimize the time to match them [Cazenave 1996a].

After machine compilation; some new knowledge is available. This new knowledge is added to the system's knowledge. The system is also able to forget previously learned knowledge if it becomes useless and harmful because it takes time to unify the corresponding rules when their conclusions are already deduced by other rules. It verifies that the new rules added is more general than some of the previously learned rules (It does it using unifications between first order rules), if it is the case, it destroys the old specific rules. This is the way it forgets the useless and memory consuming old knowledge. Another forgetting mechanism is a filter that is applied before integration of rules to the system's knowledge. This filter is used to avoid the utility problem of learned rules [Minton 1988]. The filter consists in metarules that tells which rules are harmful. For example, it systematically forgets the rules that conclude on a set of forced moves which has a cardinality greater than five. This is because forced moves are used at AND nodes in the proof trees developed during games, a AND node with more than 5 branches has good chances to fail. The other reasons is that the bigger the list, the more conditions are to be fulfilled, so these rules are the ones that are the most likely to fail and to add match time without being applied. This is why it forgets them.

Introspect is able to anticipate the consequences of moves on some goals, to reason about its own knowledge and to bootstrap: it uses the knowledge it learns to learn other knowledge. The expansion is stopped by using metarules that enables it to forget rules of low utility. The trace of machine reasoning is equivalent to short term memorization. Our model is consistent with the suggestion of [Minsky 1987], in his chapter on consciousness and memory, he states that self-consciousness concerns our thinking about our recent thoughts. Introspect is equivalent to the "meta-management" layer of the overall architecture for conscious systems described in [Sloman 1997a]. In section 6, when explaining why unconsciousness can be useful, we will briefly describe the equivalent of the deliberative and reactive systems in Gogol (The Go program written by Introspect).

5 A language and a metalanguage for machine Introspection

Following the review paper by Barklund [Barklund 1994], a metalanguage is a language that can represent another language, called an object language. Introspect uses a completely declarative logic language. The order in which the rules of the programs are fired does not change the final result of the program. Introspect uses true negation, but not negation as failure in its domain theory. Meanwhile, it uses negation as failure to know that some sentence is not a consequence of some knowledge [Konolidge 1988]. An interesting property of our metalanguage is that it can represent itself, this is quite important if a system wants to reason at different metalevels. Self-reference has been studied extensively by [Perlis 1985,1988]. Other logic programming language such as Gödel [Hill & Lloyd 1994] have representations of themselves. A possible extension of such types of self-representable languages is to have a theory that represents itself, such autoepistemic theories are interesting for formalizing agents upon their knowledge or deductive capabilities [Konolidge 1988]. A lot of examples of self references can be found in [Hofstadter 1979]. Autoepistemic theories are not yet used by Introspect.
As example of some possibilities of Introspect, we give a rule and a metarule taken from our application to the multi-agents simulation. Here is a rule describing the evolution of the simulation:

Vision_angle (?n 1) :- Position_pedestrian (?n1 ?x ?y), Dx_angle (?n ?dx), Dy_angle (?n ?dy) , equal (?x1 add (?x ?dx)), equal (?y1 add (?y ?dy)), Identification_case (?n2 ?x1 ?y1), not_equal (?n2 -1), not_equal (?n1 ?n2).

This rule means that the emplacement that is one step ahead of the pedestrian with angle ?n*PI/10 cannot be occupied by the pedestrian (Vision_angle (?n 1)). This is due to the fact that the position of the pedestrian number ?n1 is at location ?x,?y (symbols with question marks are variables), and that a step in the direction of angle ?n*PI/10 would make him move at ?x+?dx,?y+?dy. Unfortunately, the number (?n2) of the emplacement at ?x+?dx,?y+?dy is not empty (not equal to -1) and not already occupied by the pedestrian (not equal to ?n1).
There are sixty rules that calculate all the predicates related to the choice of the orientation of the pedestrian in the simulation.
Example of a metarule about the monovaluation of a predicate:

replace_variable (?r ?var1 ?var4) :- rule (?r), condition (?r Identification_case (?var1 ?var2 ?var3)), condition (?r Identification_case (?var4 ?var2 ?var3)), not_the_same (?var1 ?var4).

This rule means that there is only one possible value for each emplacement in the simulation. If the system creates a rule that contains two different variables for the same emplacement, then it replaces one of the variables by the other one (?r is a variable containing a rule, ?var is a metavariable containing another variable, the metapredicate 'condition' looks for all the conditions in rule ?r that match the given predicate).
Metaprogramming in logic is a very useful tool for program manipulation, but also for controlling logic programs, for reasoning about knowledge, reasoning about reasoning. All these abilities are very useful when writing an introspective symbolic program that improves itself by reasoning on its behavior.

6 Unconsciousness can be useful

Introspect memorizes its mental actions so as to be able to observe itself after. It can transform itself into an unconscious program by compiling itself. The benefit of being unconscious is that it is faster because it does not have to interpret and memorize its behavior. The drawback is that it cannot introspect itself anymore. The unconscious program written by Introspect is named Gogol and consists of 1 000 000 lines of C++.
When working in the unconscious mode, Gogol can notice that he failed to solve a problem, but he cannot explain to itself why a move succeeded in solving the problem. However Gogol can use this information to automatically create new interesting problems for Introspect (The knowledge of Gogol is the unconscious equivalent of the knowledge of Introspect, so the same moves surprise both of them).
Gogol is not completely unconscious, he uses some self monitoring during his tree searches. He develops AND/OR tree searches so as to solve problems in games. While developing the search trees, he dynamically looks at the shape of the tree so as to choose the leaf to develop. He chooses the leaf of the tree that will prove the problem with the least work, based on the number of leaves in his subtree that still have to be proved, this is based on Proof Number search algorithm [Allis 1994].
In Gogol, there is a deliberative level that contains knowledge on search and on strategic decision. There is also a reactive level that use compiled knowledge telling what moves to consider to solve a problem. And finally, the meta-management level is very small and consists in rule that tells which moves are surprising, this level is the one that enables to call Introspect when appropriate and use a much larger meta-management level. But Introspect is not used when Gogol plays in competitions because Gogol has to answer moves of the opponents in 10 seconds.

7 Results

7.1 The tactical part of a Go program

Introspect has written the tactical part of the Gogol Go program. The tactical part of the program is the most important one. The program written by Intro-

spect enables Gogol to select between 0 and 5 moves that can achieve a specific goal. The mean number of legal moves is 250 on a Go board, it is impossible to search by looking at all the moves. So proving that out of these 250 legal moves, only between 0 and 5 are useful dramatically reduces the complexity of the search. When searching at depth n, instead of having 250^n boards to evaluate, there are only a^n boards to evaluate with $0<a<5$. In the game of Go, n is often as high as forty. The compilation in C++ enables the program to run 60 times faster than the logic program. The Go program plays a move in 10 seconds on a Pentium 133 MHz, for each move it proves about 450 tactical theorems, each theorem requires between 4 and 600 nodes in a search tree to be proved, at each node of each tree, the C++ program written by Introspect is called to find the useful moves to try.
Gogol competed in the international computer Go tournament held during IJCAI97 together with 40 other participants. It finished as the best program based on academic research, playing better that the other programs directly written by Artificial Intelligence researchers and Go professionals. It has outperformed commercial systems that have required more than 10 person*years of work.

7.2 Modelization of pedestrian in a realistic multi-agent simulation

Introspect has also been used to rewrite the decision part of a pedestrian in a commercial urban simulation. It has written a C++ program that is 5 to 10 times faster than the original C++ program written by the authors of the simulation.
Our goal is to create realistic urban simulations involving pedestrians, cars, pedestrians crossings and many others urban agents. These simulations help architectural designers in choosing architectural configurations. A problem related to this simulation is to create agents that have realistic behaviors and that are also efficient (a simulation may manage thousands of agents at the same time, so modeling an agent's behavior has to be rapid). Creating a realistic agent's behavior manually is hard because of the great number of cases and interactions that can take place. Some programmers have worked on programming manually agents behaviors during months, but some of the agents still had unrealistic behaviors, leading to unrealistic simulations. Moreover, the model was very sensitive to changes in an agent: a little and apparently unimportant change in an agent could transform a working simulation into an unrealistic one. Therefore, we have developed a program that automatically improves the agents behaviors given (1) some simple situations to avoid (a car that run over a pedestrian, or a pedestrian that tries to walk on another one) and (2) the rules of the simulation. The rules that describes the world and the rules describing the situations to avoid are written using predicate logic. The program that automatically writes the agents is written using metapredicates that manipulates the predicate logic rules describing the simulation. The metarules are in charge of writing all the possible rules that can lead to a situation to avoid in the next steps of the simulation. This enables the agents using these rules to be more realistic. The advantage of creating them automatically is to have a lot of reliable, efficient and quickly designed rules. The creation of all the rules is made by replacing some predicates in the rules that describe the situations to avoid, with their definitions contained in the rules of the simulation. Our approach to automatic agent improvement is efficient and can be used in other contexts.
Simulating realistic agents behaviors is time consuming, especially in simulations containing thousands of agents. Another problem is that making agents more complicated and more realistic makes the maintaining of the program harder, and also makes changes in the program difficult to handle. The solution we found to overcome these two problems is to automatically create efficient and realistic agents from a declarative description of their behaviors. The goal of the method that our system optimizes is to find the move of each pedestrian in the simulation. It is called very often and it is a time consuming method.
Given the rules presented, our system wrote a C++ method that is much faster than the original method. The rapidity of the synthesized program is one advantage over the traditional programming approach. Another advantage is that it is easier to modify the behavior of an agent when it is written in a declarative logic language than when it is directly written in C++. The main reason for the success of this approach is that hand-coded programs have to be maintainable and simple so that the programmer can understand them, whereas our system does not have this limitation. The clarity of an hand-made program is sometime at the price of its efficiency. Our system writes long and unclear (for humans) programs, but they are faster than hand-coded programs because all the specializations that can be made have been made. Thus, our approach enables to write faster agents simulations, and also enables to modify agents behaviors in an easier way than by directly modifying the C++ code of the agent. It would be interesting to link this application to other work on synthesis of agents [Petta & al. 1997] [Sloman 1997b].

7.3 A model with a lot of applications

Introspect has been used in many domains (games, pedestrian simulation and management) and has discovered, by introspection, knowledge that is more efficient than the knowledge given by experts. The methods used in Introspect, creating efficient program by self-observation can be applied in many different domains.

8 Future work

Applying the system to itself has partially being done and has given encouraging results. Learning to learn [Schmidhuber 1994] and its parallel in our system :

being conscious of its own consciousness is an exciting area of development of our approach. Our current research is about specializing and changing the representation of the rules of the game by reflecting on the efficiency of its introspective learning.
In [Trappl & al. 1997], inductive machine learning and Case-based Reasoning are considered for preventing the outbreak of wars or for ending them. We believe that negotiations and compromises between countries can be modeled as an abstract game. Using our learning system for this game seems a promising application. Our system is a kind of deductive learning system, so it would complete well the scope of machine learning methods used on this problem. We want to generalize our approach by applying it to many other domains. The prevention of war domain, and the synthetic agent domain seem to be promising domains of application of introspective learning methods.
In [Sloman 1997a], some characteristics that conscious systems should have are described, we tried to analyze Introspect using these characteristics. What Introspect does : learn things, takes decisions, make plans, consider options, compare things, makes inferences, notice processes and relationships, classify things, forget things, feels puzzled, switching attention, can get happy/unhappy or envious (when playing Go). What Introspect does not (yet): having new sensory experiences, becoming angry or relieved, rehearsing arguments, reminiscing, forming attachment, acquire new tastes, having a new impulse to act or think in a certain way.

9 Conclusion

Introspect is a (meta)system that builds better than human systems in complex and well defined domains. Introspect is a first step in the direction of having a more general model of consciousness and learning. It has the merit of being a running system that demonstrate the usefulness and feasibility of a kind of machine self-consciousness. Its great success is to be able to create by introspection, and given the rules of the game, a Go program that is better than some commercial Go program that have required more than 10 person*years of human consciousness and work. Moreover, it is a general system that has given similar results in other domains. Creating implemented models of consciousness is a promising way of rapidly increasing the intelligence of machines. This work is also a step towards integrating philosophical concepts and AI programs [Sloman 1995].

References

[Allis 1994] - L. V. Allis. *Searching for Solutions in Games and Artificial Intelligence*, Ph.D. Thesis, Vrije Universitat Amsterdam, Maastricht, 1994.

[Barklund 1994] - J. Barklund. *Metaprogramming in Logic*. Encyclopedia of Computer Science and Technology, eds. Allen Kent & James G. Williams, Marcell Dekker, New York, 1994.

[Bouzy 1995] - B. Bouzy. *Modélisation Cognitive du Joueur de Go*. Ph.D. Thesis, Université Pierre et Marie Curie, Paris 6, 1995.

[Cazenave 1996a] - T. Cazenave. *Automatic Ordering of Predicates by Metarules.* Metareasoning and Metaprogramming in Logic Workshop, Bonn, 1996.

[Cazenave 1996b] - T. Cazenave. *Système d'Apprentissage par Auto-Observation. Application au Jeu de Go.* Ph.D. Thesis, Université Pierre et Marie Curie, Paris 6, 1996.

[Dennett 1991] - D. C. Dennett. *Consciousness explained.* Penguin Press, Allen Lane, 1991.

[Hill and Lloyd 1994] - P. M. Hill, J. W. Lloyd. *The Gödel Programming Language.* MIT Press, Cambridge, Mass., 1994.

[Hosdtadter 1979] - D. Hofstadter. *Gödel, Escher, Bach : an Eternal Golden Braid.* The Harvester Press, Hassocks, 1979.

[Konolidge 1988] - K. Konolidge. *Reasoning by Introspection.* in: P. Maes and D. Nardi (eds.), Meta-Level Architectures and Reflection, North-Holland, Amsterdam, 1988.

[McCarthy 1996] - J. McCarthy. *Making Robots Conscious of their Mental States*, in Muggleton S., editor, Machine Intelligence 15, Oxford University Press, 1996.

[Minton 1988] - S. Minton. *Quantitative results concerning the utility of Explanation-Based Learning*, AAAI88, p. 564-569, 1988.

[Minsky 1987] - M. L. Minsky. *The Society of Mind*, William Heinemann Ltd., London, 1987.

[Perlis. 1985] - D. Perlis. *Languages with Self-References I: Foundations*, Artificial Intelligence, 25:301-322, 1985.

[Perlis. 1988] - D. Perlis. *Languages with Self-References II: Knowledge, Belief and Modality*, Artificial Intelligence, 34:179-212, 1988.

[Petta & al. 1997] - P. Petta, R. Trappl. *Why to Create Personalities for Synthetic Actors*, in : Trappl R., Petta P. (eds) Creating Personalities for Synthetic Actors, 1997.

[Pitrat 1990] - J. Pitrat. *Métaconnaissance*, Editions Hermes, Paris 1990.

[Robson 1983] - J. M. Robson. *The Complexity of Go* - Proceedings IFIP - pp. 413-417 - 1983.

[Schmidhuber 1994] - J. Schmidhuber . *On Learning how to Learn Learning Strategies*, TR FKI-198-94, Technische Universität München, 1994.

[Sloman 1995] - A. Sloman. *A Philosophical Encounter*, IJCAI 1995, Montreal, 1995.

[Sloman 1996] - A. Sloman. *Functionalism*, Newsgroup sci.psychology.consciousness, 22 Feb. 1996.

[Sloman 1997a] - A. Sloman. *The Evolution of What?*, http://www.cs.bham.ac.uk/~axs, 1997.

[Sloman 1997b] - A. Sloman. *What sort of Control System is Able to Have a Personality ?*, in : Trappl R., Petta P. (eds) Creating Personalities for Synthetic Actors, 1997.

[Trappl & al. 1997] - R. Trappl, J. Fürnkranz, J. Petrak, J. Bercovitch. *Machine Learning and Case-based Reasoning : Their Potential Role in Preventing the Outbreaks of Wars or in Ending Them*, OEFAI-TR-97-10, 1997.