

Infinite Regress with Self-Monitoring

Sylvie Kornman

LAMSADE Université Paris Dauphine

Place du Marechal de Lattre de Tassigny

75775 Paris cedex 16 , France

e-mail kornman@lamsade.dauphine.fr

Abstract

To exhibit some autonomy and to be able to treat unexpected situations, a system must have the possibility to monitor its own behavior: it must be able to assess, analyze and possibly fix its own behavior. A natural extension to this approach is to give the monitoring system the possibility to apply upon itself. Some researchers have claimed that this is impossible because it would lead to infinite regress.

The claim of this paper is that a sufficient condition for a self-monitoring system to avoid infinite regress is that its knowledge is sufficient for tackling the monitoring system possible misbehaviors. We call this condition “reflective self-sufficiency”. On the one hand, this approach eliminates possible infinite regress. On the other hand, this has the advantage of reducing the amount of knowledge required by the system.

We support our claim by describing the experiments conducted with the SADE system. SADE has been designed as a testbench for meta-level architectures. In its present version, the main concern of SADE's monitoring is to prevent both looping and stopping, which is a minimum necessary condition for survival. We show from various examples how to implement reflective self-sufficiency in this case.

1. Introduction

To exhibit some autonomy and to be able to treat unexpected situations, a system must have the possibility to monitor its own behavior. Psychologists' studies pointed out that human problem solving includes good monitoring [Chi 89], [Anderson 89], [Schœnfeld 85]. Indeed, a common belief is that problem solving depends on general cognitive abilities that can be applied to a wide range of domains [Holyoak 91]. This hypothesis provides a theory of problem solving where problem solving is assimilated to search [Polya 57], [Newell 72]. In this view, a problem is solved by decomposing its goal into subgoals when no operator is available for solving the problem, or by selecting the adequate operator. The problem then relies on selecting the appropriate branch to pursue or the appropriate operator to apply. This approach leads to what is known as the control problem.

In some cases, analyzing the problem features makes it possible to find the right direction to pursue. However, in most cases, this does not immediately provide a solution, and then selection is based on heuristics. Even if heuristics are usually good for focusing the search towards promising directions, they may fail in unusual cases. Thus, the heuristic nature of problem solving can lead search to wrong directions. So, controlling problem solving must be supplemented by monitoring. This is true for human problem solvers as for computational problem solvers: intelligent systems may face

problems where decisions must be made with no standard criteria for validity. For this reason, intelligent systems have to be able to monitor their own actions: they must be able to assess, analyze and possibly fix their own behavior.

A natural extension to this approach is to give the monitoring system the possibility to apply upon itself. Nevertheless, as noticed by Hofstadter [Hofstadter 85]:

“This seems prime territory for an infinite regress: an endless hierarchy of structures each one monitoring changes in the level below it.”

In this case, a system exhibits just the opposite of autonomy: it apparently stops acting because internally climbing the infinite tower.

The claim of this paper is that a necessary condition for a self-monitoring system to avoid infinite regress is that its knowledge is sufficient for tackling the monitoring system possible misbehaviors. We call this condition “reflective self-sufficiency”. Indeed, even if the monitoring system cannot be deal with all situations that may occur at the basic level, since unexpected situations may occur, it can be made capable of dealing with all malfunctions that it may suffer from, just because they are known at design time. Moreover, this approach has the advantage of reducing the amount of knowledge required by the system, and, if the system’s monitoring scope is sufficiently wide, of making it capable of solving a large class of problems, even unpredictable ones, while it can be made robust in its design by just coping with monitoring possible trouble.

We support our claim by describing the experiments conducted with the SADE system. SADE is a monitoring system that has been designed as a testbench for meta-level architectures. In its present version, the single concern of SADE’s monitoring is the detection and correction of loops and dead-ends. We have decided to select this particular aspect of monitoring because it is a minimum, elementary condition for survival.

As SADE must be able to deal with all sorts of loops and dead-ends, SADE’s knowledge is general, but rough and incomplete. So, SADE may fail in monitoring and need to be monitored. The monitoring system can be applied to itself, leading to any number of meta-levels and thus, to infinite regress. SADE has been turned into a reflective self-sufficient system by introducing monitoring knowledge drawn from previous experiments. We show through various examples how this could be achieved here.

The rest of the paper is structured as follows: Section 2 presents the SADE system, Section 3 shows which knowledge has been introduced into SADE in order to avoid monitoring failures at a second monitoring level. Section 4 presents related work, and Section 5 concludes.

2. The SADE system

In this section we present the SADE system. SADE is designed to detect and correct loopings and dead-ends. For sake of clarity, we will just expose how the SADE system deals with loopings, and leave aspects involved by dead-ends (for the second aspect, see [Kornman 93]). We first give an overview of SADE's monitoring process. Then, we present an example of SADE's operation. Finally, we give an overview of SADE's knowledge.

2.1. The monitoring process

SADE monitors a basic knowledge base which is interpreted by an inference engine. Strategic choices involve the choice of the operator to be examined and the choice of variable bindings. For each type of choice, the inference engine has several methods at its disposal. The role of the monitoring system is to dynamically determine which method to use in a specific case. When starting, the inference engine uses default control methods. Usually, they consist of selecting the first operator available and binding a variable to the first value available. Those methods are both time and memory cost-effective. In particular, they do not require memorizing previous firings, and no effort is wasted on decision-making process. It is clear that, in some cases, they may lead to looping.

SADE's monitoring process is consistent with psychologists' argument according to which monitoring consists of the periodic evaluation and assessment of solutions as they evolve, and the curtailing of attempts that are unfavorably assessed. It can be summarized as follows.

SADE suspects the occurrence of a looping whenever the number of basic level firings has reached some fixed threshold. Then, it tries to confirm or to invalidate the looping. It interrupts basic level inference, selects a number of firings to be scanned next, changes the trace mode, and fixes a later date, when basic level inference will be analyzed. Then, basic inference resumes. At the previously settled date, basic inference is interrupted, and SADE analyzes the trace. If an occurrence of looping is confirmed, SADE selects an operator involved in the loop and applies a remedy to this operator during some fixed time. It settles another interruption date, when remedy effects are checked. This process is applied until the suspected loop is invalidated or disappears.

2.2. An example of SADE's operation

This section gives an example of SADE's operation. The basic level consists of operators that a robot can apply while it is situated in a labyrinth. A labyrinth is a set of rooms. Figure 1 shows a labyrinth with six rooms.

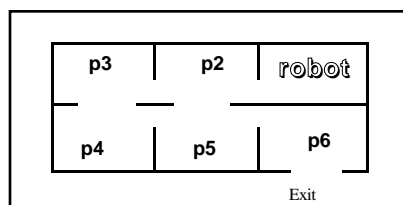


Figure 1: a simple labyrinth

The robot does not know the labyrinth configuration. It has just a local vision. It knows which room it is in and which rooms open into it. It can walk from one room to another inside the labyrinth.

The robot may reach the exit very quickly, just as it may loop forever. The simplest loop it may enter into has length 2: the robot alternately moves between rooms p1 and room p2 back and forth. Indeed, when the robot stands in room p2, it can move to rooms p1, p3 and p5. However, it has no domain knowledge to help it in its choice. The problem consists here of determining the new room to move into. This cannot be based on the labyrinth configuration since the robot knows nothing about it.

The robot navigation within the labyrinth is simulated by an operator, namely *Operator Walk*:

Operator Walk

```
If      IsInRoom (robot, ?DepartureRoom)
        ?DepartureRoom <> exit
        OpensIntoRoom (?DepartureRoom, ?ArrivalRoom)
Then    Kill IsInRoom (robot,?DepartureRoom)
        IsInRoom (robot, ?ArrivalRoom)
```

Operator Walk only expresses the possible moves of the robot, but not next room to move into. This choice is left to control.

The looping between rooms p1 to p2 is caused by the following firings of *Operator Walk*:

First firing

(?DepartureRoom/p1 and ?ArrivalRoom/p2)

Second firing

(?DepartureRoom/p2 and ?ArrivalRoom/p1)

and so on.

The expression of *Operator Walk* is not responsible for the loop, but the way the inference engine processes it. Here, the loop depends on the choice of the value which instantiates the variable ?ArrivalRoom. As mentioned above, the default method for choosing the value to be bound to a variable consists of selecting the first value available. This is why the choice for instantiating the variable ?ArrivalRoom is always the same.

SADE's operation on previous looping is as follows:

Detection

The first stage consists of detecting the loop. SADE suspects the occurrence of a loop because the number of firings at the basic level exceeds the arbitrary fixed threshold.

Trace modification

Up to now, there is no trace. SADE decides to examine the next firings closer. It selects the number of firings to be scanned, changes the basic trace level, and fixes a date when the new trace will be scanned.

Basic level

Basic inference resumes. Operator applications, variable bindings, and the maximum number of idle choices are now memorized in the trace.

Trace analysis

At the previously fixed time, SADE interrupts the basic firings. Considering last firings trace, it confirms the basic loop and imputes it to *Operator Walk*.

Remedy choice and application

In order to repair the basic loop, SADE selects a remedy among the remedies to repair loops, and decides to apply it to *Operator Walk*. SADE selects the remedy called *DelayBindings*. *Remedy DelayBindings* changes the method for choosing the variable bindings of the operator it applies to. This method consists of memorizing all variable bindings used by the operator, and of selecting those variable bindings that have never been used yet – instead of the first ones available –. Before giving hand to the basic level, SADE fixes a new interruption date, at the end of which remedy effects are to be checked.

Basic level

The new method enables the robot to visit new rooms. The basic level firings now bring the robot from room p1 to room p2, for no other choice is available. Then the robot moves from room p2 to room p3, from room p3 to room p4 and from room p4 to room p5. At this time, the new method use ends off.

Observation of the remedy effects

SADE interrupts the basic level firings again. It analyzes the trace of the last firings, notices that the basic loop is fixed, and stops monitoring the basic loop.

Basic level

Basic level inference resumes again – with no trace –, and takes the robot to the labyrinth exit.

This example is characteristic of the SADE process. Next paragraph presents how SADE's process is implemented in SADE's knowledge base.

2.3. SADE's knowledge base

This section presents some elements of SADE's knowledge base. Other elements that make SADE reflectively self-sufficient will be presented in Section 3.

2.3.1 Looping analysis

SADE's knowledge base includes an analysis component called to confirm looping. It may return three different results, according to the scanned firings. First, it may invalidate the suspected looping. Second, it may confirm the looping. If the loop involves idle choices, the looping is classified as *PossibleChoices*. In the contrary, it is clear that every alternative has been tried and that no solution exists. So, the looping is classified as *NoSolution*.

2.3.2 Remedies against looping

SADE's knowledge is general, while incomplete. It includes four general remedies to correct looping. Three of them consist of temporarily applying control methods, different from the default ones, to a selected operator. The fourth one, *StopInference*, stops the inference. First and second remedies modify the usual control method for choosing the value to be bound to a variable. The third remedy modifies the default control method for choosing the operator to fire. First and second remedies, namely *DelayBindings* and *ForbidBindings*, both consist of memorizing all variable bindings used by the operator, and of selecting those variable bindings that have never been used – instead of the first ones available –. The difference is that the first one allows past bindings to be used when no new ones are available, while the second one never allows old bindings. *Remedy DelayBindings* allows the use of past variable bindings, while *Remedy ForbidBindings* just forbids them. The third remedy, *DelayOperator*, consists of delaying some selected operator that takes part in a looping. As will be shown later, first and third remedies may have no effect on a loop, while the second one always stops looping, but may stuck inference.

A remedy is applied during some fixed time, sufficient enough to take the monitored level out of a loop. So, *Remedy ForbidBindings* and *Remedy DelayBindings* lengths are the maximum between the double of loop length and the maximum number of idle variable bindings. Likewise, *Remedy DelayOperator* length is the maximum between the double length of the loop and the maximum number of idle operators. This makes the monitored level able to try the maximum of idle choices in order to get out of a loop.

Each remedy is a black box that can repair a certain kind of looping. SADE includes little knowledge to make the right remedy selection. SADE selects *Remedy StopInference* when the looping is classified as *NoSolution*. However, in other cases, remedy selection depends on the order remedy descriptions appear in SADE's data structure. This is why, after having applied a remedy, SADE must observe its effects. If a looping continues or if the firings are stuck, SADE still tries to apply a remedy until the looping disappears.

2.3.3 SADE's operators

SADE's monitoring process is controlled by a decision component that selects the action to execute. It includes 4 operators. Each operator corresponds to a specific stage of the monitoring process.

Operator LengthChoice is fired when the looping is not yet confirmed. It asks for an analysis, and selects a number of firings to scan.

Operator RemedyChoice is fired when the looping is confirmed, but not yet repaired. It selects both a remedy and an operator involved in the loop, and asks for the application of the selected remedy to the selected operator.

Operator StopMonitoring1 stops the present monitoring process, when the looping is repaired.

Finally, *Operator StopMonitoring2* stops the present monitoring process, when the looping is invalidated.

Those 4 operators are mutually exclusive. This implies that only one of them can be selected at each step. So, strategic choices only involve the number of firings to scan, the remedy to apply, and the operator a remedy should be applied to.

2.3.4 SADE's limits

SADE cannot confirm any looping, for a loop length cannot be known in advance. SADE's knowledge includes several possible lengths that might be selected by *Operator LengthChoice*. The present maximum is 1000. It means that loops longer than 1000 firings cannot be confirmed. However, the inference engine stops inference after 10 000 basic firings.

Next section shows why and how SADE must and can be applied at a second monitoring level.

3. Monitoring the monitoring system

The need to set up a second monitoring level comes from SADE's incapability to deal with all situations that may occur at the basic level. SADE's knowledge is not sufficient to analyze and classify any looping it has in charge. However SADE's knowledge can be made sufficient to let it detect and correct SADE's own failure, for such malfunctions are known at design time. We call this condition reflective self-sufficiency.

SADE may fail in three different ways. The monitored level is looping, but SADE cannot confirm it. SADE detects a loop and selects a remedy with no effect, while there is a remedy that can correct the loop. SADE detects a loop and selects a remedy with no effect, while there is no remedy to correct the loop among the available remedies.

In the rest of section, we examine one by one each kind of failure.

From now on, M1, and M2 respectively denote the first, and the second monitoring levels, while M0 denotes the basic level.

3.1. Unconfirmed looping

SADE is unable to confirm a looping it has detected when the selected number corresponding to the firings number to scan is lower than the loop length.

To examine this case, we go back to the robot looping inside the labyrinth. We now assume that the robot's loop is 20 firings long and that M1 decides to scan 10 firings in order to confirm the basic looping. Then, after 10 basic firings, M1 cannot confirm the basic looping and fixes a new date, still 10 firings later in order to scan next 10 firings. In such a case, M1 starts looping by always repeating the same actions:

```
Scanning last 10 firings without results.  
Selecting number 10 of firings to scan  
Fixing a date, 10 firings later
```

Now, M0 and M1 are both looping. M2 will now correct M1's looping and make it able to correct M0's looping. M2's operations are as follows:

Detection

M2 detects M1's looping.

Modification of the trace

M2 changes the trace level of M1's firings and fixes a later date, in 10 firings.

Level M1

M1's inference goes on exactly as before.

Trace analysis

At the previously fixed date, M2 interrupts M1's firings. M2 analyzes the trace of last 10 M1's firings, and confirms M1's looping. M2 imputes the looping to *Operator LengthChoice*, because the only selection point, involved in the looping, is the selection of the number of firings to scan, made by *Operator LengthChoice*

Remedy choice and application

Both remedies *ForbidBindings* and *DelayBindings* can correct M1's looping, while *Remedy DelayOperator* has no effect on it. As shown in next Section, *Remedy DelayOperator* cannot be selected here. We assume that M2 selects *Remedy DelayBindings*. This remedy will make *Operator LengthChoice* try all possible numbers of firings to scan, until M1 confirms the M0's loop. Before giving hand back to M1, M2 fixes the next interruption date.

Level M1

M1's inference resumes. The method for choosing variable bindings of *LengthChoice* now consists of choosing first new variable bindings instead of the first ones available. M1 must now choose a number different from 10. Assume that M1 chooses 25.

Level M1

At the end, M1 confirms the basic looping and corrects it, as exposed in Section 2.

This example shows how M1 may fail and loop when it cannot confirm a looping. It also shows how M2 can detect and repair this failure. In order to prevent such kind of failure at level M2, “10” is the minimum firings number that might be selected. As it will be clear after we will have finished exposing all situations of SADE’s failure, it turns out that “10” is longer than the length of any loop that may occur at level M1. This ensures that if M1 ever loops, M2 is able to confirm it immediately. So, the first kind of SADE’s failure cannot occur at level M2.

3.2. A wrong remedy selection

We now assume that M1 has properly detected a M0’s loop, but is unable to select a proper remedy, while one exists. We present which knowledge has been introduced into SADE in order to prevent such kind of failure.

As already mentioned, remedy selection depends on the order remedies descriptions appear in SADE’s data structure. However, the quality of SADE’s monitoring must not depend on this order. Moreover, if SADE’s knowledge includes the right remedy to repair a loop, SADE must be able to select and apply that remedy, even if it does not select it at first. In previous example, *DelayBindings* was the first one stored, and thus the first one selected. To show a wrong remedy selection, we assume that remedies are stored in order *DelayOperator*, *DelayBindings*, and *ForbidBindings*, and we go back to the robot inside the labyrinth.

We now assume that the robot navigation is simulated by four operators: *North*, *South*, *West* and *East*, with the obvious meaning. When starting, the robot is in room p1 (Figure 1) and loops by alternately moving from room p1 to room p2. The looping is caused by the alternate firings of operators *West* and *East*.

The basic looping can be repaired by applying to *Operator East* any of both remedies *DelayOperator* and *ForbidBindings*. The application of *Remedy DelayOperator* to *Operator West* has no effect, just as *DelayBindings* on *East* or *West*. Finally, the application of *Remedy ForbidBindings* to *Operator West* sticks the robot in room p1. Indeed, in this case, *Operator West* can lead the robot from room p1 to room p2. Then, *Operator East* can lead it back to room p1. Back to p1, the only possibility is to apply *Operator West* that takes the robot from room p1 to p2. However, this move has already been used, and is now forbidden by *Remedy ForbidBindings*.

Here, M1 detects M0’s looping, confirms it, and imputes it to both operators *West* and *East*. Then, it decides to apply *Remedy DelayOperator* to *Operator West*. As this has no effect, M1 notices that the basic looping continues, and then keeps applying *Remedy DelayOperator* to *Operator West*. So, M1 comes into the loop consisting of always repeating the same actions:

```
Noticing that M0’s looping is not repaired
Applying remedy DelayOperator to Operator West
```

The first action comes from the component that observes remedies effects, when the second one comes from *Operator RemedyChoice*. The only selection node involved in the loop is the selection of values bound to both remedy and operator selected by *Operator RemedyChoice*.

M2 detects and confirms M1's looping, and must select a remedy to repair it. M1's looping can be repaired by the application of both remedies *DelayBindings* and *ForbidBindings* to *RemedyChoice*, while *DelayOperator* has no effect. Indeed, both remedies *DelayBindings* and *ForbidBindings* make M1 select a new remedy, and so stops M1's looping, while *Remedy DelayOperator* cannot affect M1's looping, because M1's loop includes no node of operator choice. So, should M2 select *Remedy DelayOperator* to correct M1's looping, it would start looping, exactly like M1, by always repeating:

```
Noticing that M1's looping is not repaired
Applying remedy DelayOperator to RemedyChoice
```

Here, M2 would require the intervention of a third monitoring level, and so on. So, we have here a possible occurrence of infinite regress.

To prevent M2 from selecting *Remedy DelayOperator*, we have introduced into SADE knowledge that forbids selecting *Remedy DelayOperator* when the idle operators list of the loop is empty. In the example above, this knowledge prevents M2 from selecting *Remedy DelayOperator*. It is not applicable here to M1, because the idle operators list of the loop includes *South* and *West*. Indeed, when the robot stands in room p2 both operators are applicable.

Thus, basic looping is repaired here by the joint action of M1 and M2:

First, M1 selects *Remedy DelayOperator* on *Operator West* to repair the basic looping and so, gets looping.

Second, M2 selects *Remedy DelayBindings* to repair M1's looping.

Finally, M1 selects *Remedy DelayOperator* on *Operator West*.

Basic looping is fixed.

Let's make some remarks.

First, notice that *Remedy DelayOperator* is selected a second time, for it is still the first element in SADE's data structure. Therefore, it is now applied to *Operator West* instead of *Operator East*. Second, *Remedy DelayBindings* application makes M1 select all possible remedies until it finds out a right one.

The knowledge introduced is sufficient, because the fact that SADE's operators are mutually exclusive implies that SADE's idle operators list is always empty. Moreover, SADE's process only includes points of value choices concerning the number of firings to scan or concerning the remedy to apply and the operator to be treated.

3.3. No remedy

We now assume that M1 has properly detected a loop in M0, but that there is no available remedy to correct it. Here, inference must be stopped, because the looping cannot be repaired.

We shall also take the advantage of this example to outline how applying SADE to itself has an influence in the way SADE's knowledge is expressed. We expose first how the expected behavior is obtained when M1 is monitored, and then how it would be obtained with only one monitoring level.

3.3.1. M1 is under surveillance

In the case there is no remedy to correct M0's looping, M1 and M2 first act as previously described. M1 starts looping by always applying the same remedy to the same operator. Then, M2 detects the looping, confirms it, and decides to apply *Remedy DelayBindings* to *RemedyChoice* for the necessary time, letting M1 try all combinations of remedies and operators involved in M0's looping. M1 tries all possible remaining remedies on M0's loop, without result. Therefore, M1's looping has been fixed. So, when M2 observes the effect of the remedy it has selected, it remarks that M1's looping is repaired.

If M2 just contented with this result, it would stop monitoring M1's looping here, and then M1's looping would resume, leading one more time to M2's intervention, and so on. M2 would also start looping, and need the intervention of a third monitoring level, while M2's looping cannot be repaired. In order to avoid that failure, we introduced knowledge making the observation component check the number of idle variable bindings choices. If this number is 0, the observation component sends the result *NoSolution* back. So, in such case, M2 selects *Remedy StopInference*, because M1's looping cannot be repaired.

Notice that, the same result would have been obtained, if M2 had selected *Remedy ForbidBindings* instead of *Remedy DelayBindings*.

3.3.2. M1 is not under surveillance

Assume now that there is no second monitoring level. Then, the expected behavior can be obtained by adding into SADE's decision component an operator looking like:

StopFiring

If There is a malfunction
And All the remedies to correct it have been tried
And The malfunction is not repaired
Then Stop the firings

Solutions with or without self-monitoring both provide the same behavior. However, in the case M1 is under surveillance, knowledge introduced into the system is two-fold: First, it may be used by M2 in cases like the one exposed. Second, it may be used by M1 when the basic level is looping, looking for a solution which does not exist.

In conclusion, this example shows how infinite regress is prevented when M1 detects a loop that no remedy can repair. Moreover, it shows that the use of a second monitoring level reduces the amount of knowledge required by the system.

In this section, we have presented SADE's failure, and shown what knowledge has been introduced into the system to prevent these failures at level M2. This knowledge is limited and does not tell when and when not to select a specific piece of knowledge. It would have been insufficient, if only used at one single monitoring level. Therefore, it is sufficient, when used at two monitoring levels, the first level monitoring the basic level, and the second level monitoring the first monitoring level.

4. Related Work

Dynamic reflection potentially leads to infinite regress. Several approaches have been proposed to address that problem in other domains than monitoring, [Smith 82], [Laird 87]. They are all based on a common idea: there must exist a default, compiled and safe mechanism, that, at some level, substitutes for more clever but less sure knowledge.

In systems like 3-Lisp [Smith 82] this infinite regress is terminated by having a second non-recursive architecture whose effect is the same as an infinite number of unmodified meta-levels. This architecture can then be used instead of the recursive architecture when it would have the same effect. The approach taken by Soar [Laird 87] is similar. Soar has been initialized with a set of productions that provide a complete default architecture for situations in which there is no specific knowledge about how to proceed. These productions prevent the actual generation of infinite, context-free, meta-level hierarchies by generating preferences for a default response when a subgoal occurs for which no problem space is suggested.

The solution kept with the SADE system has consisted of starting with the same domain-independent default architecture at both levels M1 and M2. While this default architecture cannot be considered as safe and sure by itself, both levels M1 and M2 provide a two-level architecture with maximum security SADE's knowledge can offer.

5. Conclusion

In this paper, we have discussed the problem of self-monitoring in relation with the problem of autonomy. We have argued that a sufficient condition for a self-monitoring system to reach some autonomy is that its knowledge can tackle its own monitoring misbehaviors. This condition is necessary to avoid infinite regress and is sufficient to ensure good monitoring. As shown with the SADE system, it can be realized in a practical way by introducing the right knowledge into a system. The scope of SADE's rational knowledge is limited, but sufficient, while used at two meta levels. This implies that limited rationality can lead to more rationality when applied at different meta levels.

6. References

- [Anderson 89] J.Anderson, F Conrad, A.Corbett, "Skill Acquisition and the LISP Tutor", *Cognitive Science*, Volume 13, 1989, pp 467-505
- [Batali 88] Batali J., "Reasoning about Self-Control, Meta-Level Architectures and Reflection", P. Maes and D. Nardi editors, 1988, pp 255-270
- [Chi 89] M.T.H.Chi, M.Bassok, M.W.Lewis, P.Reiman, R.Glaser, "Self explanations : How Students Study and Use Examples in Learning to Solve Problems", *Cognitive Science*, Volume 13, 1989, pp 145-182

- [Holyoak 91] K.J Holyoak, "Problem Solving, An Invitation to Cognitive Science", Volume 3, *D.N Osherson ed*, pp 117-146
- [Hofstadter 85] Hofstadter D., "Metamagical Themas : Questions for the Essence of Mind and Pattern", 1985, Basic Books
- [Kornman 93] S.Kornman, "SADE : un Système Réflexif de Surveillance à Base de Connaissances", thèse de l'Université Paris 6
- [Laird 87] J.E.Laird, A. Newel and P.S Rosenbloom, "SOAR : An Architecture for General Intelligence", *Artificial Intelligence*, Volume 33, 1987, pp 1-64
- [Maes 87] P.Maes, "Computational Reflection", PhD Dissertation, Vrije Universiteit Brussel,
- [Pitrat 90] Pitrat J., "Métaconnaissance, futur de l'intelligence artificielle", Hermès
- [Polya 57] G. Polya, "How to solve it", *Garden City*, 1957, NY: Doubleday/Anchor
- [Schœnfeld 85] A.H.Schœnfeld, "Mathematical Problem Solving", Academic Press
- [Smith 82] Smith B.C., "Reflection and Semantics in a Procedural Language", PhD Thesis, TR272, CS Lab, MIT, 1982.
- [Smith 86] Smith B.C., "Varieties of Self-Reference", Proceedings of 1986 Conference on Philosophical aspects of Reasoning about Knowledge. Los Altos, CA : Morgan Kaufman, 1986, pp 19-43