Meta-Knowledge, Autonomy, and (Artificial) Evolution: Some Lessons Learnt So Far

Jean-Luc Dormoy
EDF R&D Center
IMA-TIEM
1, avenue du Général De Gaulle
92141 Clamart Cedex, France

Sylvie Kornman
LAFORIA
Paris 6 University
4, place Jussieu
75252 Paris Cedex 05, France

Abstract

We claim in this paper that an extensive use of meta-mechanisms is a very powerful tool for building autonomous AI & AL systems. We support that claim by examples of *knowledge-based systems* exhibiting unexpected and partly autonomous behaviors. They show that autonomy, as well as viability, could be achieved in the future by means of meta-mechanisms. In particular, meta-mechanisms require simpler mechanisms than multi-agent-like or emergent mechanisms, though they can achieve more sophisticated behaviors.

We also present the perspective of bootstrapping as a basic methodology for building AI & AL systems. Its motivation is that building AI or AL systems is probably too difficult a problem to be tackled by just finding out the right components and then assembling them. These systems must go through a sequence of evolutionary steps, and we must be aware of that fact.

On one-hand, bootstrapping means that any new component should be made *applicable to the system as a whole* - and not only *participate* to the system's behavior - in order to help further extensions. On the other hand, the strategy used when building a system should *not* be mainly concerned by immediate performance, *but* by further extensions. This means that new functions or components should be added to the system *in order to* make further extensions possible or easier. We describe some experiments in building AI systems according to this strategy, and its constraints and difficulties.

Indeed, we have no clean theory of bootstrapping yet, instead mere intuitions. So, these experiments mainly aim at figuring out how the design of a sophisticated system through bootstrapping steps should be conducted. We think that, eventually, a theory will be required.

1 Introduction

A common view of AL and AI systems architectures is based on *multi-agent architectures*. In this view, a set of non-intelligent small agents interact with each other, and from this interaction emerges a global behavior. In a practical way, the physical implementation of the agents share and act through a common substrate -e.g. data structures in 'symbolic' multi-agent systems, or numbers in neural nets- , but they are of a different nature. Their own substrate, i.e. their physical implementation, does not intersect the substrate they act on.

The aim of this paper is to discuss the meta view, i.e. a view where agents can also *act on* agents. This view is not new in any way, but we think it can be useful to reconsider it with respect to refurbished problems stressed by the AL community, such as autonomy, viability and evolution. We make various claims, and we show how they are justified by some examples of previous AI systems, and by the systems we are building, namely the Shal and Sade/Meta-Hari systems.

We first claim that meta is powerful. Indeed, a simple mechanism achieves more when properly used at the meta-level rather than at the basic level. Acting at the meta-level means that the components implementing the mechanism act on the physical implementation of other components, whereas acting at the basic level means merely interacting with them. Conversely, the designer who intends to build a system exhibiting a given behavior would better choose a meta-level architecture.

Secondly, we re-consider the autonomy and viability problems with respect to meta-level architectures. In particular, an operational closure of the system[1] can be achieved by simpler mechanisms in a meta architecture. Autonomy can be enhanced in meta-level architectures by the fact that components can be used at any level. Indeed, from the designer's point of view, this a saving principle: let components act on components, and you will need less, because a component can participate to various functions at various levels. Viability can be tackled by specific-purpose components, which for example observe other components, and try to fix improper behaviors.

Thirdly, we discuss the bootstrapping problem of AI or AL systems. Bootstrapping has been a long-known technique for building compilers, for example. It consists of using an already-existing version of a system for running a new version. When systems have the capability of processing themselves, the old version can

[1] In the sense of Varela

often be discarded after having been used once. So, bootstrapping is a design methodology for building artifacts through a sequence of small steps. Its main benefit is that one very quickly takes advantage of the improved functions provided by the intermediate versions, instead of having to wait for the system to be completed.

Indeed, we think that bootstrapping usefulness is two-fold. First, it helps in building an operationally closed system. This means that the system is not yet fully autonomous, that it still requires the designer to intervene in its functioning, but that already built-in components can be subtituted for the designer's intervention in solving sub-problems. This is the stage we are in in the Shal construction. Second, bootstrapping could be used to improve -from the designer's point of view- an already autonomous system. It can even be thought of including 'mutation components' in the system, which would help the designer to change the system. We are not experimenting this yet.

The bootstrapping perspective is perfectly coherent with the meta view. Indeed, we try to build components which participate to their own processing. Besides, building a system in this way dramatically changes the content of the components to be implemented, i.e. what they do. When adding a new component, its function must be to help to process components, including itself. So, bootstrapping makes the designer face a new set of constraints, he is not free to tackle the subproblem he would like to. Indeed, those constraints turned out to be so tough in our experiments that they required our main focus of attention. Eventually, our systems are not intended to provide any kind of 'external' functionality, but to be merely able to process themselves. The bootstrapping methodology we used -*how*- became a goal -*what*- in its own. We describe some aspects of this work.

Let's say at last, and at least, some words about the examples provided throughout this paper. First, we felt free to borrow from famous pieces of work in AI, and to stress some of their aspects which seemed interesting to us. This does not mean that their authors share our views. Second, our examples are *knowledge-based systems*. We know the discussion about *representation*. Anyway, we think that these systems properly illustrate our views, though not specific to knowledge-based systems. Instead, we think that, at this level of discussion, using knowledge or not is a mere issue of substrate. Beyond, we also have some ideas about the previously mentioned discussion, but we do not intend to argue about them in this paper.

2 Meta is powerful

The Prodigy system is the first example we mention, which shows that a mechanism can provide a more powerful behavior when properly used at the meta-level rather than at the basic level.

Prodigy (Minton & al. 1989) is a workbench developed at Carnegie-Melone University for experimenting various kinds of learning in knowledge-based systems. Among these experiments, Steve Minton used the Explanation-based Learning (EBL) technique for learning new control knowledge which helps a problem-solver to solve a class of problems. Some form of EBL was already used 15 years ago in planners such as STRIPS. Since then, EBL has been formalized and used in various ways. The spirit of EBL is to learn from a single example and a theory of the domain (knowledge) some tractable formulations of a generally intractable *target concept*. For example, the rules of chess could be the domain theory, a particular winning chess position the training example, and the target concept a formalization of *win*.

Usually, EBL is used for learning from success. This means that, when a particular case of the class of problems is solved, EBL attempts to summarize how this success has been reached, and to generalize as much as possible. It produces some kind of *macro-operator*, which sums up the generalized version of the actions performed to reach success in the example, and which can be used later to directly solve a similar problem. The main problem in this approach is that the cost of matching previously learnt macro-operators can overwhelm the potential improvement they can provide. This is the cost/utility problem.

One of the innovative features of Prodigy is that, instead of using EBL at the basic level (i.e., the level of the problem itself), it is used at the meta-level of the problem-solver. Prodigy has been provided with a theory of some important aspects of the problem-solver behavior, and with target concepts such as success, failure and goal interaction[1]. What Prodigy learns by means of EBL is control rules, i.e. rules which guide the problem-solver's decisions when attempting to solve a problem. The experiments conducted by Minton & al. showed in particular that learning from failure actually was an effective learning method.

Beyond the intrinsic interest of Prodigy, the point we wish to emphasize is that the basic technique used in it is a "slight" modification of an already-known technique. The main difference is that the EBL mechanism is used in Prodigy at the meta-level, i.e. the level of the problem-solver. We think that that is one of the main reasons why Prodigy succeeded, while its predecessors were stuck by the cost/utility problem. Moreover, Prodigy can learn more general knowledge. For example, if a particular problem mentions the two subgoals *Stack x on y* and *Stack y on z*, then Prodigy will learn (from goal interaction) and then use the fact that it is better to try to stack y on z first. This control rule can be used in various situations, while a macro-operator learnt by EBL

[1] A goal interaction occurs in a planning problem whenever an already-achieved subgoal must be destroyed for achieving another subgoal.

when applied at the basic level would consist of a more specific piece of knowledge.

3 Meta in the perspective of autonomous and viable systems

3.1 Autonomy

The use of components at the meta-level, i.e. acting on components, makes it simpler to build relatively autonomous system. We shall give two examples, from two systems: Shal and Sade/Meta-Hari.

In the Shal system (Dormoy 1990, 1991), which the first author is currently developing, some components are used at different levels and so participate to different functions. For example, we are developping a TMS-like component[1], which function is to undo the deductions drawn from a fact later considered as non pertinent. The initial reason why we implemented this component was that Shal makes heuristic deductions in its various activities, and they can be found to be incorrect later. So, these deductions are to be undone. However, Shal also has a component which aims at discovering errors in a knowledge-base. Obviously, this error-finding component is applied to the Shal system itself (including the error component). It turns out that the error-finding component knowledge also contains heuristics. So, it happens that some errors are incorrectly stated by this component. When the system finds evidence of this, the TMS component acts on the error component - which acts on the whole system - to undo its wrong deductions. In the next future, it might also be possible that the TMS component undo deductions drawn by the TMS component (because it also has heuristic knowledge).

This example shows that various functions can be achieved by a single component, provided that it has the possibility to apply at various levels. Without systematically giving components the ability to act on components, we would have had to design specific components for each usage of our TMS component. So, it is clear to us that reaching a relative autonomy is made easier by using meta capabilities, simply because less components are required.

Another example is the Sade/Meta-Hari under developement by the second author (Kornman 1989, 1991). This system is designed to observe a running knowledge-base, to discover misbehaviors in it, such as looping or getting stuck, and to repair the knowledge-base behavior in order to get out from the wrong situation. This system has the ability to do so in various situations, but not always. In particular, it can loop or get stuck. It turns out that, together with the system, a small component for systematically interrupting a knowledge-base while it is running has been implemented. This component routinely triggers the observing system, which attempts to find evidence of a misbehavior, if any. This interrupting component applies to any running knowledge-base, in particular to the observing system when it is running. So, the system also sometimes "observes itself". When it is looping or getting stuck, a copy of itself can analyze the situation, discover the wrong behavior, and repair it. Obviously, it is possible that the system at meta-level 2, i.e. the system observing the system which observes the knowledge-base, go into looping or getting stuck. Another level of observation is then added.

It is clear that this meta-tower of mutually observing systems is not a panacea. Firstly, it is not desirable to have a high tower: while being observed, a system does nothing, and so a relatively small amount of time should be devoted to observing. Secondly, it is possible that the "whole tower" loop or get stuck: indeed, more and more levels are added, the upper levels observing the lower ones, and each level going into wrong behavior[2].

However, the experiments being conducted with the Sade/Meta-Hari system show that it is possible to design the system so that it avoids complete collapse in most situations. Roughly speaking, the observing system must not have too many problems. If so, the meta-observing system would have much work repairing its lower copy, and, as its capabilities are the same, it would not be good enough to do so. Other problems arose, which we shall just mention here, such that the relationship between form and content of a component. When we say that a component acts on a component, we should indeed say that the *content* (what it does) of a component acts on the *form* (its physical implementation) of a component. However, for a given content - which is what the designer aims at when building the component -, there are various possible forms for implementing it. So, the content of meta-components acting on components strongly depends on the form of the components they act on. In a practical way, a very slight change in the form of a component can invalidate the meta-components. So, one of the most challenging problems when building meta-level architectures is to "tune" the form of components in order to make it fit with the content of already-existing components which are to be used at the meta-level. This is still a "black art".

The Sade/Meta-Hari system is autonomous is some sense, and brings autonomy to the knowledge-base it is applied to. When coupled to a knowledge-base, it makes it possible to almost always get out from traps and deadends - even traps caused by the implementation of the procedural inference engine which runs the knowledge-bases. So, there are two lessons learnt by these experiments. First, autonomy is not reached by a knowledge base - or any other kind of system - per se,

---

[1] Indeed, this component has the function of a conventional TMS, but does not work at all in the same way.

[2] From a logical standpoint, deciding whether a program loops is an undecidable problem.

even if this was intended by the knowledge base designer. It is strongly enhanced by another component acting at the meta-level, the Sade/Meta-Hari system. Secondly, the observing component also has the capability to apply onto itself, which still enhances the behavior of the whole system.

3.2 Viability

Viability looks like a very difficult problem, even more difficult than autonomy. In the (short) history of meta-level systems in AI, the systems which exhibited a relatively autonomous behavior also exhibited a very poor ability to avoid quick collapsing. We shall first give two examples which demonstrate this problem, then an example which shows how unstability could be fought.

The AM system (Lenat 1982), designed by Doug Lenat in the 70s, was an "artificial mathematician". It had knowledge and heuristics to build and consider interesting mathematical objects, and to conjecture interesting theorems about them. It proved nothing, and the knowledge in mathematics it started from was close to nil. Simply, it exhibited the "inspired behavior" of a mathematician while discovering new mathematics. This system had stunning results. For example, it discovered basic arithmetics, including integers, addition and multiplication, prime numbers, and unique factorization of integers in primes.

However, according to Lenat, AM did not produce interesting results after running two hours. It had given anything it could, and was lost in overwhelming uninteresting objects and conjectures.

Lenat interpreted this problem in the following way: AM had the "right heuristics" to deal with the simple objects it started with, but once more sophisticated objects were introduced (arithmetics), these heuristics were of no use any longer. AM lacked the ability to synthesize new heuristics better fitting its new domains of interest.

This was the main motivation for Lenat building his next system, Eurisko (Lenat 1983). Eurisko had heuristics to deal with heuristics, in particular heuristics for changing or discovering new heuristics. The results of Eurisko have also been quite impressive. It managed to win a naval battle game championship, by being trained to learn good heuristics specific to this game. It also reproduced the results of AM, and others in other domains (e.g. VLSI design). In particular, Lenat showed that Eurisko had discovered a new meta-heuristic, i.e. a heuristic for discovering heuristics, which was better than the heuristic which gave birth to it. This was an example that the system could actually enhance itself.

However, it is not clear how Lenat could manage to keep his system relatively stable. Obviously, a system which can radically change itself has many opportunities to produce lethal components. This actually happened in Eurisko. For example, a meta-heuristic had been synthesized, which stated that nothing in the system was interesting. This "killing" component fortunately turned out to be suicidal, i.e., while killing everything, it killed itself. But this was just chance, and it seems reasonable that more subtle wrong components have been generated. It seems that Lenat managed this problem by often intervening in the Eurisko process of discovery (he says that Eurisko's successes are 60% Eurisko's and 40% Lenat's). So Eurisko was not *viable* by itself, it required Lenat's help. There was something lacking in Eurisko, but this thing is obviously sophisticated.

We shall now show how the viability problem could be tackled by using a rather different example. The Shal system we are developping makes a systematic use of its knowledge for processing itself and helping its own design process. A very serious problem when building such a sophisticated system is *errors*. This is true for any system, but systems based on a meta-level architecture can exhibit some new kind of errors, much more difficult to identify and fix than in other systems. When a component is faulty, and when this component applies to another component, the fault can be visible only a long time after. For example, if a component MC1 -used at the meta-level - makes deductions on a component C which helps another component MC2 to compile C, and if an error occurs in MC1, then the error can be visible only when using the compiled version of C. But, in turn, component MC1 is not necessary faulty, simply we used a compiled version of it, wherein an error has been introduced by the compiling components. So, tracking down an error requires not only to observe the behavior of the interacting components, but also to navigate through the meta-levels of components acting on components.

The simplest idea for tackling this problem is to design a special component, the aim of which is to find errors in components. In a practical way, this requires the discipline which consists, whenever the designer (us) finds out a new kind of error, of providing the system with sufficient knowledge to discover similar errors on its own the next time they occur. We have in Shal an ever-growing error-finding component, which size now adds up to as much as one-third of the whole system.

It is rather difficult to quantitatively assess the gain of this error component. However, we experienced that it does discover errors when we modify Shal, and our past experience convinces us that it helps to save days of work.

This error component does not exactly respond to the viability problem as usually stated, in the sense that it function is not to modify the behavior of the system when something lethal is coming up, but to prevent the system from being so wrong that it would not "survive" more than a few minutes. However, if we think of the design process of Shal as some kind of ontogenic process, the error component prevents from generating wrong components.

Another example is the already-mentioned Sade/Meta-Hari system. While the error component of Shal could only statically analyze a system, Sade/Meta-Hari dynamically intervenes in the behavior of the

system. So, this system provides a partial answer to the viability problem as generally understood.

Both systems show, though still in a partial way, that the use of components at the meta-level can help to tackle the viability problem. Moreover, this problem can be *explicitely* tackled. The fact that a system's behavior is viable does not only emerge from its architecture, instead there are some specific components which help it to do so. The problem is to discover some relatively general mechanisms which ensure that a system will not collapse.

4 Bootstrapping

4.1 Methodologies for building AI & AL systems

In Nature, there are several degrees of change. Usually, AL people emphasize the adaptive aspect of autonomous systems. This refers to the way "grown-up" animals or "completed" artificial systems behave. If we refer now to Artificial Life or Artificial Intelligence systems, it is clear that these systems must have a degree of adaptivity, but an even more difficult issue is to know how to build them - adaptive or not. In Nature, this is done by reproduction, and evolution, which are very different degrees of change.

A consistent view in AI is the "explosive kernel" vision. Instead of building a huge system by hand, we should try to build some very special components, which should be able to improve themselves just by running, so providing us with a new, better system. These primordial components make up the kernel, and its intention is to be explosive, i.e. to expand and improve itself in an infinite loop.

A serious experiment in this way was Lenat's Eurisko system. Eurisko was provided with meta-heuristics, more precisely heuristics which aimed at discovering new heuristics. Eurisko exhibited one case of a group of meta-heuristics which discovered new, better meta-heuristics. So, this first experiment was a success in showing that it is possible to build a system which intrinsically improves itself, not only adapts itself to external conditions. However, as visible at once, this perspective is extremely ambitious, and Eurisko did not go further along this line.

Another view is the bootstrapping vision. Indeed, we are seeking to build artificial systems, i.e. built by a designer, though life has appeared without any. Now, building AL and AI systems is so difficult a task that it is probably hopeless to do it in one single step. So, the idea is to use what is already built as soon as it is available while building the system. This implies that the main function of components should not be to achieve an 'external' function, but to participate to the whole running and changing of the system.

This is no new idea. It has been used for long in computer science, for building compilers or interpreters (partial evaluation) for example. It has also been used in some AI systems, such as TEIREISIAS (Davis 1982). TEIREISIAS, which was a system put upon the MYCIN expert system, was aimed at explaining and controlling MYCIN's behavior, and at helping the expert to acquire new knowledge. The TEIREISIAS component dealing with acquiring new objects, attributes and values had been built by Randall Davis by means of itself: as soon as some primitive concepts had been wired-in, the object, attribute and value concepts themselves were acquired by means of TEIREISIAS.

Jacques Pitrat has put the idea forward in his MACISTE system (Pitrat 1986, 1990). The main -and only- aim of this system is to be able to process itself. Roughly speaking, the system is a rule-based system. It has a very important component, which is a rule compiler. One of the main goal of the MACISTE experiments was to show that an AI system could produce all the programs it would need -not a single line of code should be written by the human designer. So, the problem is to build a rule compiler which can compile itself. In a practical way, this is no easy task.

Starting from Jacques Pitrat's ideas, we are currently building a system, named Shal, which is also intended to fully process itself. Nevertheless, there are some differencies between Shal and MACISTE, due to different starting points. We shall not discuss them, neither shall we discuss the details of these systems. Instead, we shall describe some lessons learnt so far in building an operationally closed system through bootstrapping.

4.2 Constraints and problems in bootstrapping

4.2.1 The bootstrapping tunnel: make steps small

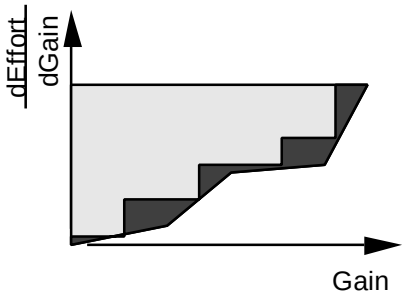

Figure 1: The bootstrapping tunnel

Our first attempt consisted of directly building a rule compiler written in rules. This work had been done with Jean-Yves Lucas in 1988-89 (Lucas & Dormoy 1990). We started from his SIREN system, which was a system able to synthesize programs which solved constraint satisfaction problems from their mere specification. This system had 400 rules, and was theoretically able to compile its rules. However, the experiments showed that rules compiled by this compiler were less efficient than when interpreted by the human-encoded inference engine, especially when run on large working memories. There was some knowledge implicitely present in the

inference engine which was not stated within the rules. So, we considered adding rules. We assessed that the system's rules would add up to 800. However, when compiling rules R, the compiling rules work on a representation of R in the working memory. So, when represented, 800 rules make up a large working memory, which was just the case when our first compiler was inefficient. So, we got stuck.

The reason why this first attempt was a failure was that we tried to achieve a sort of operationally closed system through a single step. This is not a good idea, just because we could not take advantage of an already-existing system. This is summarized in the diagram below. The area under the lines respectively represent the effort (e.g. human, or CPU) required by achieving a given function when completing the system through a single step (grey), or through a sequence of small steps (dark). The reason why the second alternative is better is that each intermediate system takes advantage of the functionalities provided by the previous ones.

4.2.2 The necessity to solve various problems at the same time

After this first attempt, we chose another path. Instead of trying to reach a rule compiler very quickly, we added some knowledge for discovering the behavior of a rule base. The idea was that this would provide better compilers in the long term, and at the same time that 'partial' conclusions drawn by this component could be immediately useful for improving the rules behavior and efficiency. This is what actually happened. Though not yet able to compile rules, Shal takes advantage from what is deduced by Shal to improve its behavior.

Then, other problems happened. The Shal knowledge base was getting ever larger, and we had many problems in simply managing this large ensemble. First, a better control was required for the system properly using its knowledge. We designed a language and some knowledge for stating, using, and 'criticizing' control. Second, from the beginning, we had been using *semantic constraints*, which provided the system with knowledge about the kind of working memory to be used. It turned out that, as the system grew larger, it was not possible to provide it with this kind of knowledge by hand. So, some components were added, which make it possible to discover this knowledge automatically.

While adding these components, new 'micro-languages' had appeared, wherein we -or the system- stated semantic constraints or control knowledge. All this knowledge was represented within the working memory, in a very painful-to-read syntax. So, we designed a component for translating simple languages into each other from simple specifications.

Secondly, more and more knowledge was of a heuristic nature. This means that it turns to be somewhat 'irrational', and that at least some means for checking and correcting them was needed. This is why we implemented a 'TMS-like' component -though very different in nature from what has been done in AI about this topic.

Thirdly, the most important obstacle became *errors*. There are various kinds of errors. First, there are the errors in implementing the system. Second, when using heuristic knowledge, the system can go into 'wronger and wronger' deductions, and so collapse or enter any kind of lethal behavior. For both reasons, the largest component in our system is now an error-finding-and-fixing component. It adds up now to one third of the whole system.

Eventually, this very short chronology of our system development shows that it is not possible to solve a problem independently from others. One must tackle 'all' the problems at the same time. Seeking autonomy and viability, we are now quite far away from mere rule compiling.

4.2.3 The necessity to solve problems in a bad way

At the same time, we have had a hard time to get rid of old psychological and cultutal habits. In particular, when a scientist faces a problem, he tries to solve it *in-depth*. This does not work here. Indeed, if for a given subproblem P, we implement an excellent solution in a component C, we can expect C to be large. As C must be processed by other components, these components must be 'expert enough' to cope with that large one. In the first stages of our system development, this is not possible. So, components must remain relatively small and have a form as simple as possible.

4.3.4 Dealing with the "real-world" I: the fixed point problem

Adding a component is two-fold. One first wants to add a new functionality; this is its *content*, i.e. what the component is supposed to do. Now, this component must be processable by already-existing components. Indeed, when acting at the meta-level, components act upon the *form* of components. There is no simple relationship between content and form of a component. Even a slight change in the 'language' where components are expressed -or, say, their physical substrate- can dramatically change the form of a component exhibiting a given behavior.

So, the designer is not free to add components. When adding a new component C1, with an intended content, there must be somewhere what is required to process the form of C1. If this does not exist yet, one must add C2 for processing C1. In turn, one might need C3 for processing C2, and so forth. At the end, we must have in hand components C1, C2,…,Cn, which content can process their form. This is some kind of 'fixed point' property, or 'local operational closure'. In a practical way, the size of C1 to Cn must not be so large as to make them unprocessable by the already-existing system (see Subsection *The Bootstrapping Problem*).

We have no answer to this problem, but vague intuitions. Indeed, this is the main problem we are fighting with in building Shal.

4.2.5 Dealing with the "real-world" II: internal vs. external worlds

In some sense, the Shal and Sade/Meta-Hari systems are perfectly egocentric: they just observe themselves, compile themselves, process themselves, etc. In particular, they are not confronted with the real world. Their 'real world' is themselves.

Indeed, this is not a side-effect of our approach, but one of its foundations. As we mentioned, designing components, the content of which is able to process their form, is a difficult task. So, adding components for making the system able to achieve an intended behavior in the 'real world', let alone 'surviving' in it, is out of reach of our systems in their current stage of development.

We think that this is not a problem, on the contrary we think that this might be a necessary condition for long-term research. Our main goal is not performance, it is 'artificial evolution'. As we said, we think that bootstrapping, or 'artificial evolution', could be an efficient methodology for building working AL & AI systems. So, trying to reach a quick solution when building an autonomous creature might be a deadend in the long term. We do not mean that there is nothing to learn from these experiments. We are on a different research path. We are seeking new design principles for AL & AI systems. A consequence of our approach is that our systems will go on surviving in 'protected worlds' for a long time.

5 Conclusion

We showed in this paper that meta-level architectures make simple mechanisms more efficient when used at the meta-level. We also showed that autonomy and viability can be enhanced in the meta approach. We then introduced the bootstrapping, or 'artificial evolution' approach, which aims at building a system by means of its earlier versions. We described some conclusions learnt so far from our experiments in building the Shal and Sade/Meta-Hari systems. What we omitted to mention is that we have no clean theory yet, but we think that such a theory will be necessary in the future.

References

Davis, Randall, and Doug Lenat. 1982. *Knowledge-based systems in Artificial Intelligence.* Mc Graw-Hill, 1982.

Dormoy, Jean-Luc. 1990. *Behavior and function of a knowledge-base*. Cognitiva'90, November 1990, Madrid, Spain.

Dormoy Jean-Luc. 1991. *Knowledge for compiling knowledge: the Shal system*. To be published in the *Revue de l'Intelligence Artificielle.*

Kornman, Sylvie. 1989. *Automatic introspection in a declarative knowledge-base system.* Congrès systémique, Lausanne, Switzerland.

Kornman, Sylvie. 1991. *Systems under surveillance.* IASTED 1991.

Lenat & Davis. 1982. See Davis, 1982.

Lenat, Douglas B. 1983. *EURISKO: A program that learns new heuristics and domain concepts*. Artificial Intelligence 21 (1983), pp. 61-98.

Lucas J-Y, and J-L Dormoy. 1990. Jean-Yves Lucas, Jean-Luc Dormoy. *Shal : un compilateur de règles écrit en règles qui s'applique à lui-même (A rule compiler which applies to itself).* Convention IA'90, January 1990, Paris, France.

Maes, Pattie, and Daniele Nardi, Eds. 1988. *Meta-level architectures and reflection.* North-Holland.

Minton, Steven, Jaime G. Carbonell, Craig A. Knoblock, Daniel R. Kuokka, Oren Etzioni, and Yolanda Gil. 1989. *Explanation-based learning: a problem-solving perspective*. AI Journal Vol. 40, Numbers 1-3, September 1989.

Pitrat, Jacques. 1986. *Le problème du bootstrap*. (*The bootstrapping problem*). Internal report "Cahiers du Laforia".

Pitrat, Jacques. 1990. *Métaconnaissance.* (*Metaknowledge*). Hermès, Paris.

Varela, Francisco J. 1989. *Autonomie et connaissance, essai sur le vivant.* Seuil, Paris.