Journal of Artificial General Intelligence 3(3) 1-10, 2012
DOI: 10.2478/v10229-011-0017-1

Submitted 2012-12-19
Accepted 2011-12-19

Editorial: Approaches and Assumptions of Self-Programming in Achieving Artificial General Intelligence

Kristinn R. Thórisson THORISSON@RU.IS
Center for Analysis & Design of Intelligent Agents /
School of Computer Science
Reykjavik University
101 Reykjavik, Iceland
&
Icelandic Institute for Intelligent Machines
2.h. Uranus
101 Reykjavik, Iceland

Eric Nivel ERIC@RU.IS
Center for Analysis & Design of Intelligent Agents /
School of Computer Science
Reykjavik University
101 Reykjavik, Iceland

Ricardo Sanz RICARDO.SANZ@UPM.ES
Autonomous Systems Laboratory
Universidad Politécnica de Madrid
José Gutiérrez Abascal 2
28006 Madrid, Spain

Pei Wang PEI.WANG@TEMPLE.EDU
Department of Computer and Information Sciences,
Temple University
Philadelphia, PA 19122, USA

1. A Working Definition of Self-Programming

Intuitively speaking, "self-programming" means the ability for a computer system to program its own actions. This notion is clearly related to Artificial Intelligence, and has been used by many researchers. Like many other high-level concepts, however, scrutiny shows that the term can be interpreted in several different ways. To make the discussion concrete and meaningful we introduce here a working definition of self-programming. In this definition we increase its concreteness while trying to keep the intuitive meaning of the concept.

The activities of a computer system usually are considered to consist of atomic **actions** (which can also be called ***instructions***, ***operations, behavior***, or something else in different contexts). At any given moment the system's primitive actions are in a finite and constant set ***A***, meaning that they are distinct from each other, and can be enumerated. An action may take some *input arguments*, and produce some *output arguments*. The system can execute each of its actions, as long as certain *conditions* are satisfied. When executed, an action produces certain *effects* in the system and the outside world.

A **program** refers to a structure of actions to be executed *(a)* in the future (a "plan" plus "contingencies"), *(b)* currently being executed (an "activity"), or *(c)* already executed in the past (a "record" or "memory"). A program's actions – its parts – are executable by the system, while data are defined as non-executable parts. A program accepts some input data, and produces some output data and side effects. Actions can be related to each other according to their order of execution, or their sharing of data or arguments. The system's activities or behaviors can thus be seen as the execution of one or multiple programs. Any instantiation of any part of the system is in one of two states – or roles: either a data item or an action. An instance of a data item can become executable at a particular point in time, and vice versa. Likewise, an executable item can *be turned into* a data item, but again, it will not serve both roles simultaneously without duplication. Given a reflective programming language (such as Lisp or Python) – a programming language can treat its own program elements as data – a system written in it can modify its own source code to become a different program.

A computer system usually can be described at different levels. At the bottom level, the executable actions, as well as the programs constructed out of the actions, are determined by the hardware, which can only accept a certain finite set of instructions, usually fairly small. However, execution of programs can be taken as actions, too. Furthermore, some programs (such as compilers and interpreters) translate programs from one language into another language. Therefore, a program at one level can be data at another level (i.e., where it cannot be executed). Even so, for a given level (usually called a *virtual machine*), the action set $\boldsymbol{A}$ is still determined, which in turn determines what is considered as programs and data on that level. Therefore, in the following, "actions" and "programs" are always used with respect to a certain virtual machine.

By **self-programming** we mean the production of one or more programs created by the system *itself*, whose principles for creation were provided to the system at design time, but whose details were decided by the system at runtime based on its experience. In other words, the self-generated program is determined by some factors in the interaction between the system and its environment. In this process, the action set $\boldsymbol{A}$ may still be fixed, but the executed programs are not predetermined by the system's designers. Concretely speaking, self-programming is the process carried out by the system that takes a set of actions and a goal as input, and produces a program composed of the actions as output, which is expected to satisfy the goal.

We can further distinguish self-programming into levels, according to the "depth" to which this process penetrates and affects the system. *Level one* self-programming capability is the ability of a system to make programs that exclusively make use of its primitive actions from action set $\boldsymbol{A}$. *Level two* self-programming systems can do this, and additionally generate new primitives. Level three self-programming adds the ability to change the principles by which Level one and Level two operate, in other words, Level three self-programming systems are capable of what we would here call **meta-programming**. This would involve changing or replacing some or all of the programs provided to the system at design time. Of course, the generations of primitives and the changes of principles are also controlled by some programs. Though the process of self-programming can be carried out in more than one level, eventually the regress will stop at a certain level. The more levels are involved, the more flexible the system will be, though at the same time it will be less stable and more complicated to be analyzed.

The concept of self-programming as explained above is related to artificial intelligence in the following way: As far as it is the goal of an artificial system to improve its performance on some task or set of tasks, a self-programming system should – in theory – be better equipped at improving its performance on a task, as it has the ability to change itself in more fundamental ways than a system designed in the traditional way, where it is fully specified beforehand by a human programmer. Nevertheless, as the self-programming needs to be guided by another (meta) program, the fact that a system has the ability to program itself is not a *guarantee* that it is in a better position than a traditional system. In fact, it is in a worse situation because in this case there are more ways in which its performance can go wrong. Without a powerful set of principles to guide the self-programming the system will therefore not reach its goal faster or better than any other program, quite possibly the opposite will happen. However, the inherent limitations of hand-coding methods make traditional manual programming approaches unlikely to reach a level of a human-grade *generally intelligent system*, simply because to be able to adapt to a wide range of tasks, situations, and domains, a system must be able to modify itself in more fundamental ways than a traditional software system is capable of. Ultimately, a hand-crafted system operating in an environment that is prone to change in unpredictable ways from what it was when the system was designed – no matter how minor – is likely to fail in unpredictable ways.

For AGI the set of relevant self-programming approaches shall be a much smaller set than that typically discussed in computer science, and in all likelihood strongly linked to what we generally think of as the architectural structure of such systems, since self-programming for AGI may fundamentally have to change, modify or partly duplicate, some aspect of the architecture of the system, for the purpose of being better equipped to perform some task or set of tasks.

2. Existing Approaches to Self-Programming

According to the above working definition of self-programming the following AI techniques can be seen as approaches toward self-programming. Though their assumptions and solutions are very different, they all aim at organizing actions into a *program* to *achieve a given goal*, without following a predetermined algorithm that is specialized at the goal.

Since each of the techniques comes with many variants, and they can be combined, the following addresses their most common forms.

[S] State-space search (example: GPS (Newell 1963). The atomic actions are state-changing operators, and a program is represented as a path from the initial state to a final state. Variants of this approach include *program search* (examples: Gödel Machine (Schmidhuber 2006)): Given the action set ***A***, in principle all programs formed by it can be exhaustively listed and evaluated to find an optimal one according to certain criteria.

[P] Production system (example: SOAR (Laird 1987)). Each production rule specifies the condition for a sequence of actions that correspond to a program. Mechanisms that produce new production rules, such as chunking, can be considered self-programming.

[R] Reinforcement learning (example: AIXI (Hutter 2007)). When an action of an agent changes the state of the environment, and each state has a reward value associated, a program corresponds to a *policy* in reinforcement learning. When the state transition function is probabilistic, this becomes a Markov decision process.

[G] Genetic programming (example: Koza's Invention Machine (Koza et al. 2000). A program is formed from the system's actions, initially randomly but subsequently via genetic operators over the best performers from prior solutions, possibly by using the output of some actions as input of some other actions. An evolution process provides a utility function that is used to select the best programs, and the process is repeated.

[I] Inductive logic programming (c.f. Muggleton 1994). A program is a statement with a procedural interpretation, which can be learned from given positive and negative examples, plus background knowledge.

Besides the above fairly well-known AI approaches, we add here two less known ones that have recently become relevant in the context of self-programming:

[E] Evidential reasoning (example: NARS (Wang 2006)). A program is a statement with a procedural interpretation, and it can be learned using multi-strategy (ampliative) uncertain reasoning. The details of this approach is described in the article by Wang in this issue.

[A] Autocatalysis (example: Ikon Flux (Nivel 2007)). In this context the architecture is in large part comprised of a large collection of models, acting as hierarchically organized controllers, executed through a contextually-informed, continuous auto-catalytic process. New models are produced automatically, based on experience, their quality evaluated in light of this experience, and improvements produced as a result. Self-programming occurs at two levels: The lower one is concerned with performance in a set of domains, making models of how best to achieve goals in the external world at any point in time, the higher level is concerned with the operation of the lower one, implementing integrated cognitive control and meta-learning capabilities. Semantically closed auto-catalytic processes maintain the system's growth after they are deployed.

There are also some related techniques, which are not considered as self-programming, according to our working definition:

- **Machine learning** in general. Though any learning inevitably changes the system's behaviors, it is not considered as self-programming if the process does not directly address how to form executable programs from a given set of actions. For example, a learning mechanism may change some parameters in a program, and in a sense the program is not the same anymore. However, this process does not compose a program from actions. To take an example, the learning algorithm C4.5 produces a program for classifying, and potentially acting on, a set of observed inputs. But as C4.5 is neither semantically nor operationally closed with respect to its execution environment: it cannot automatically adapt to changes in its environment that produce input that goes outside of the hard limited boundaries of what the human programmer set it up to operate on.
- **Automatic programming** is the technique to generate executable code from high-level specification. In essence, although the specification is more abstract than the resulting executable code, the full operational scope of the system is contained in the specification. It is not considered as *self*-programming because the program must match the specification, so the process in question is in principle like compiling or interpreting a program in a high-level language into a low-level one, rather than composing a program from actions. Model-driven development is a paradigmatic example of this approach to program construction (Balmelli 2006).
- **Reactive systems** (example: subsumption (Brooks 1986)). Each behavior of the robot is triggered by the specified condition of the behavior, and a program is the emergent behavior sequence triggered by the environment. Since such a system does not generate reusable programs, it is not considered as self-programming here.

3. Major design decisions and their assumptions

Now we can compare the approaches of self-programming in their major assumptions, and the arguments behind these decisions. Especially, we will see which assumptions can be satisfied in the context of AGI.

3.1. *How does the system represent each basic action?*

a) As an operator that transforms a state to another state, either deterministically or probably, and goal as state to be reached [R, S]
b) As a function that maps some input arguments to some output arguments [G]
c) As a realizable statement with preconditions and consequences [A, E, I, P]

Relevant assumptions: Is the knowledge about an action complete and certain? Is the action set discrete and finite?

3.2. *Can a program be used as an action in other programs?*

a) Yes, programs can be built recursively [A, E, G, I]
b) No, a program can only contain basic actions [R, S, P]

Relevant assumptions: Do the programs and actions form a hierarchy? Can these recursions have closed loops?

3.3. *How does the system represent goals?*

a) As states to be reached [S]
b) As values to be optimized [G, R]
c) As statements to be realized [E, P, A]
d) As functions to be approximated [I]

Relevant assumptions: Is the knowledge about goals complete? Is the knowledge about goals certain? Can all the goals be reached with a concrete action set?

3.4. *Are there derived goals?*

a) Yes, and they are logically dependent to the original goals [I, S, P]
b) Yes, and they may become logically independent to the original goals [A, E]
c) No, all goals are given or innate [G, R]

Relevant assumptions: Are the goals constant or variable? Are the goals externally imposed or internally generated?

3.5. *Can the system learn new knowledge about actions and goals?*

a) Yes, and the learning process normally converges [G, I, R]
b) Yes, and the learning process may not converge [A, E, P]
c) No, all the knowledge are given or innate [S]

Relevant assumptions: Are the goals constant or variable? Are the actions constant or variable?

3.6. *How much resources are demanded?*

a) Unlimited time and/or space [I, R, S, P]

b) Limited time and space [A, E, G]

Relevant assumption: Are the resources used an attribute of the problem, or of the solution?

3.7. *When is the quality of a program evaluated?*

a) After execution, according to its actual contribution [G]
b) Before execution, according to its definition or historical record [I, S, P]
c) Both of the above [A, E, R]

Relevant assumption: Are adaptation and prediction necessary?

4. **Discussion**

For real-world applications of self-programming systems, all these questions are relevant to any AGI-aspiring system. None of the concepts are Boolean in nature: They represent dimensions along which a system can be moved far, or only a short distance. Therefore it is may be inappropriate to say that they are "necessary" for AGI, one should rather adopt the stance that they enable the "G" in "AGI" to a greater or lesser extent. But at the present stage, the most important questions in our opinion may be 3.5, 3.6, and, eventually, 3.2.

The assumption about the use of resources – 3.6 – separates systems that take time as a key operating principle of the system, as opposed to systems where time of goal attainment is irrelevant. If we are considering technologies for self-programming to be used in the construction of AGI systems, the temporal performance is a critical aspect. While temporal behavior of computer-based systems has usually been considered a non-functional requirement, the late resolution of a problem may be as useless as a wrong solution, rendering the system non-functional.

The assumption about the system being able to learn new knowledge about actions and goals – 3.5 – is critical for AGI systems due to the inherent uncertainties in real-world problem solving. Initial specification of action sets and the connections between actions and goals will not be enough to cope with open-ended problems and environments as is expected from true AGI systems. The system shall be able to learn how the surrounding world behaves and, more importantly, how its own actions change world states. The causal connections between actions and goals may be a non-static set of non-static relations.

The assumption about the possibility of using a program as an action in other programs – 3.2 – may become critical not only in terms of the constructive flexibility of the system but in terms of the intrinsic robustness it may demonstrate (Hansen 2007). General intelligence systems will be characterized not only for the application of sound reasoning methods but also for their capability of achieving results that take into account their own capabilities and limitations. AGI systems shall be self-aware concerning their use and limitations of resources – both time and space as already mentioned – to be able to properly solve the problems in the finite conditions of their operational environment. Additionally they must be self-aware concerning the applicability of the knowledge they have at any specific moment and its suitability for the problem at hand. Metacognitive competences will be shown to be critical in the realization of AGI systems that can cope with their intrinsic knowledge mismatches and uncertainties – a condition that we believe every AGI-aspiring system must deal with to achieve the "G" in "AGI".

The limitations of the many and varied techniques for realizing cognitive architectures come from their assumptions, and how these assumptions are valid in specific system implementations and problem-solving episodes. Systems like NARS, for instance, that make an assumption that resources are insufficient at all times (few systems do that at present), have a head-start on self-programming, as the acquisition, organization, and use new knowledge are assumed from the outset to be managed by the system itself, after its design is complete. Adding such mechanisms to an architecture that was not designed with these assumptions is not likely to succeed, as argued by many (c.f. Thórisson 2012, Garlan et al. 1995).

With respect to AGI-aspiration, there are other capabilities – besides the capability of a system to autonomously manage acquisition, organization, and use of knowledge – that determines how far it can hope to go. We already mentioned the ability of a system to inspect its own operation: To be grounded in a domain or world a system must be able to represent itself – its own capabilities and limitations – to assess its own ability to achieve goals. Why this is critical becomes clear when we think about how a system generates sub-goals; effective generation of sub-goals can only be done in light of what the system is capable of doing, as constrained by its capabilities. And since sub-goaling is a necessary capability of a system to increase its own knowledge, modeling of self is also a prerequisite (Sanz et al. 2009).

Another capability that can bring an AGI-aspiring system forward in the AGI direction is the capacity to improve its own mental capacities. To be able to do so requires the system to observe, model, and understand its own internal operation. Essentially this is the machine equivalent of what psychology calls "cognitive growth." Cognitive growth has been proposed as a key capability for building system that can attain high levels of autonomy (Thórisson and Helgason 2012). In some sense AGI-aspiration must revolve around increased levels of autonomy, and ultimately the autonomy of a cognitive system depends on its cognitive powers, provided its embodiment can provide some minimal level of protection – or buffer (Heylighen and Joslyn 2001) – from the environment.

Cognitive growth is not possible without some form of architectural programming, and for systems capable of doing so autonomously this is a clear case of self-programming. As discussed by Thórisson (2012), architectural programming cannot be done without an understanding of the operational semantics of the architectural components, blocks, or instructions, that are being used for the programming. However, for a system that must, to a significant extent, acquire its knowledge autonomously – that is, without the hand of a human programmer – the operational semantics of the building blocks must be inferred by the system, from their observed and theoreticized operation in the world. If the blocks are very coarse-grain they automatically encompass complex internal structure, making self-inspection and self-modeling increasingly more difficult the more coarse-grain they are. To enable significant knowledge acquisition, therefore, the building blocks of an AGI architecture must be fine-grain – fine enough to provide a very basic, grounded operational semantics at the lowest level, amenable to self-inspection and self-modeling. The more fine-grain they are, the more likely it is that their semantics are transparent enough for the system to be modeled automatically. Combining many small such blocks into larger structures, or programs, can implement complex (coarse-grain) operational units without making them *black-box*. In any case, the system needs a root ontology concerning the basic types of building blocks; automatic modeling needs a core predefined set of primitive concepts. AGI system are likely to be better off concerning their own growth potential and flexibility the smaller the set of primitives and the simpler the components themselves, assuming sufficient computation is available for the system to operate at necessary speeds in its environment, other factors being equal.

Systems that make it their target to grow by principles of automatic knowledge acquisition, cognitive growth, using self-programming, are *constructivist AI* systems (Thórisson 2012). Any successful constructivist system, as we use the term, will contain a significant amount of self-generated code – most likely more than several hundred-times more than what was provided at the outset by its designers. To see why this is a logical necessity, consider the fact that any system capable of improving fundamental cognitive functions, such as e.g. knowledge acquisition, must essentially re-program the original knowledge acquisition code created by the system's designers to do so. Inevitably constructivist systems must rely heavily on self-organization as a key principle of operation.

Taking constructivist AI as the primary development methodology is thus arguably the first step towards creating systems that go significantly beyond what can be achieved by manual construction by human programmers, due to both the complexity barriers for sophisticated AGI systems, and also due to the conceptual limitations of human designers and programmers. However, there are several challenges when employing a constructivist approach. Constructivist AI may confront a problem of combinatorial explosion in relation with the many dimensions of the AGI design space. The constructivist approach requires self-programming. Self-programming can be viewed as the automatic selection of certain points of the system design space to tackle the problems and goals posed to the system. The problem of going from goals to the system structures – the programs – that are able to attain the goals, is a problem that is generally ill-posed.

5. Papers in This Special Issue

There are four articles in this special issue. Three of them address fundamental issues in self-programming and their relation to developing AGI, while one of them – by Frank van der Velde – presents a neurally-based architecture for learning linguistic constructions. He argues that in situ grounded representations can provide an important basis for AGI development, and presents a blackboard-based architecture for grounded compositional representations created from experience. With clear relevance to cognitive processes for understanding any situated experience, whether they involve language or not, van der Velde argues that linguistic competence must be part of any high-level self-organizing cognitive architecture. While not convinced that language capabilities are necessarily a prerequisite for achieving AGI, we do agree with him that language is an important part of the AGI puzzle, and most certainly an important capability for AGIs that are intended to interact with humans.

Sergio Pissanetsky's work is based on the underlying assumption that causal inference is a natural principle for the design of AGI. To this end, he presents work on what he calls a natural logic, which has resulted from his research on the self-organizing properties in canonical matrices under minimization of a certain functional. The logic generates its own semantics and presents a new type of inference, based on minimizing an action functional. Arguing that since human coders create algorithms inferred from their own experience, self-programming must be the mechanization of inferring algorithms from experience.

Wojciech Skaba has developed a cognitive architecture based on self-programming principles, called AGINAO. His paper describes the principles behind the self-programming, which is based on creating concept networks where the nodes are patterns discovered by the system. Assuming only the computation power provided by a single, present-day desktop personal computer, an implementation of the system is still in the development stage.

Pei Wang presents work on an architecture envisioned to solve problems without algorithms. His NARS non-axiomatic reasoning system, which has been under development for over a decade, assumes that AGIs will always have insufficient computational resources and knowledge for *perfect* problem solving. This fundamental principle has significant effects on the design of a cognitive architecture, especially in terms of what can be given to the system at design time, but also on operational constraints and bootstrapping of such systems.

Like Skaba and Pissanetzky, Wang's motivation for looking at self-programming stems from a need for AGIs to acquire their own knowledge, as human programmers are unlikely to be able to think of – and encode at design time – everything that the system should know during its lifetime. NARS uses logic as the foundational mechanism of knowledge representation and acquisition.

While the work presented by the four authors covers quite a breadth of topics and addresses issues of importance, it can be safely said that the research community has only barely begun to address issues related to the self-programming needs of AGIs. Similarly, significantly more work – especially systems integration – is needed before we can have clear grasp of the potential represented by the ideas presented here, not to mention allowing their use in actual implemented systems deployed in the real world, solving real-world problems. We are nevertheless convinced ideas present in these four articles will prove to be important in future efforts to develop AGI, and it will be interesting to see where the proposed research paths lead in the coming years.

References

Balmelli, L., Brown, D., Cantor, M., and Mott, M. (2006). Model-driven systems development. *IBM Systems Journal*, **45**(3):569-585.

Garlan, D., Allen, R. and Ockerbloom, J. (1995). Architectural mismatch or why it's hard to build systems out of existing parts. *Proceedings of the Seventeenth International Conference on Software Engineering*, 179-185.

Hansen, L. P. and Sargent, T. J. (2007). *Robustness*. Princeton University Press.

Heylighen, F. and Joslyn, C. (2001). Cybernetics and second order cybernetics. In R. A. Mayers (ed.), *Encyclopedia of Physical Science and Technology*, *Vol. 4,* Academic Press, 155–170.

Hutter, M. (2007) Universal algorithmic intelligence: A mathematical top →down approach. In B. Goertzel and C. Pennachin, editors, *Artificial General Intelligence*, Cognitive Technologies, pages 227–290. Springer, Berlin, 2007.

Koza, J. R., Keane, M. A., Bennett III, F. H., and Mydlowec, W. (2000). Automatic Creation of Human-Competitive Programs and Controllers by Means of Genetic Programming. In *Genetic Programming and Evolvable Machines*, **1**:121-164.

Laird, J. E., Newell, A. and Rosenbloom P. S. (1987) Soar: an architecture for general intelligence. *Artificial Intelligence*, **33**(1):1–64, 1987.

Muggleton, S. and de Raedt, L. (1994). Inductive logic programming: Theory and methods. The Journal of Logic Programming, 19–20, Supplement 1(0):629–679.

Newell, A. (1963). A guide to the general problem-solver program GPS-2-2. RAND Corporation, Santa Monica, California, Technical Report No. RM-3337-PR.

Nivel, E. (2007). Ikon Flux 2.0. Reykjavik University, School of Computer Science Technical Report, RUTR-CS07006.

Pissanetzky, S. (2011). Emergence and Self-organization in Partially Ordered Sets. *Complexity*, **17**(2):19-38.

Sanz, R., Hernández, C., Gómez, J., Bermejo-Alonso, J., Rodríguez, M., Hernando, A., and Sánchez, G. (2009). Systems, models and self-awareness: Towards architectural models of consciousness. *International Journal of Machine Consciousnes*s, **1**(2):255–279.

Schmidhuber. J. (2006). Gödel machines: Fully self-referential optimal universal self-improvers. In B. Goertzel and C. Pennachin, (eds.), *Artificial General Intelligence*, 199–226. Springer Verlag, 2006.

Thórisson, K. R. (2012). A New Constructivist AI: From Manual Construction to Self-Constructive Systems. In P. Wang and B. Goertzel (eds.), *Theoretical Foundations of Artificial General Intelligence,* Atlantis Thinking Machines, **4**:145-171.

Thórisson, K. R. & H. P. Helgason (2012). Cognitive Architectures and Autonomy: A Comparative Review. *Journal of Artificial General Intelligence*, **3**(2):1-30.