Self-Programming: Operationalizing Autonomy

Eric Nivel & Kristinn R. Thórisson

Center for Analysis and Design of Intelligent Agents / School of Computer Science, Reykjavik University
Kringlunni 1, 103 Reykjavik, Iceland
{eric, thorisson}@ru.is

Abstract

Lacking an operational definition of *autonomy* has considerably weakened the concept's impact in systems engineering. Most current "autonomous" systems are built to operate in conditions more or less fully described a priori, which is insufficient for achieving highly autonomous systems that adapt efficiently to unforeseen situations. In an effort to clarify the nature of autonomy we propose an operational definition of autonomy: a *self-programming* process. We introduce Ikon Flux, a proto-architecture for self-programming systems and we describe how it meets key requirements for the construction of such systems.

Structural Autonomy as Self-Programming

We aim at the construction of machines able to adapt to unforeseen situations in open-ended environments. *Adaptation* is used here in a strong sense as the ability of a machine not only to *maintain* but also to *improve* its utility function and so, in partially specified conditions with limited resources (including time) and knowledge. As a case in point, today's Mars rovers would simply ignore the presence of an alien character waving its tentacles in front of the cameras: observers on Earth would probably see and identify it, but the rover itself would simply not be aware of this extraordinary situation and engineers would have to upload software upgrades to change its mission and plans. In sharp contrast to such engineering, we expect adaptation to be performed automatically, i.e. with *no further intervention by programmers after a machine enters service*. Our adaptable rover would be fitted with software aiming at discovering facts in a *general* fashion, that is, not limited to ultra-specific mission schemes. This software would ideally be able to generate new missions and related skills according to the context, within the limitations of its resources - hardware, energy, time horizon, etc.

Structural Autonomy

The mainstream approach outlined above consists in the main of sophisticated ways for selecting and tuning hard-coded goals and behaviors for handling situations framed in hard-coded ontologies. Systems designed this way belong to the class of *behaviorally* autonomous systems [2], and result in fact from the traditional top-down design approach: a machine's operation is fully specified, as is the full range of its operating contexts, and it is the duty of its operator to ensure that the operational conditions always comply to said specification - otherwise the system ceases to function correctly. The point here is that such machines are meant *not to change*. Adding such change, or adaptation, to the requirements of a machine calls for an orthogonal perspective that addresses change as a desirable and controllable phenomenon. We envision motivations, goals and behaviors as being dynamically (re)constructed by the machine as a result of changes in its internal structure. This perspective - *structural* autonomy - draws on Varela's work on *operationally closed* systems [14]:

> "machine[s] organized (defined as a unity) as a network of processes of production (transformation and destruction) of components which: (i) through their interactions and transformations continuously regenerate and realize the network of processes (relations) that produced them; and (ii) constitute it (the machine) as a concrete unity in space in which they (the components) exist by specifying the topological domain of its realization as such a network."

Although this generic definition applies primarily to bio-chemical substrates, it can be adapted to computational substrates. We map Varela's terminology as follows:

- *Component*: a program. The function of a program is to synthesize (i.e. to produce or to modify) other programs. For example, generating new missions is creating new programs that define goals, resource usage policies, measurement and control procedures, etc. In this view, planning is generating programs (a plan and a program to enact it). In a similar way, learning is modifying the existing programs to perform more efficiently.
- *Process*: the execution of a program.
- *Network of processes*: the graph formed by the execution of programs, admitting each other as an input and synthesizing others (rewrite graphs).
- *Space*: the memory of the computer, holding the code of the machine, exogenous components (e.g. device drivers, libraries) and its inputs/outputs from/to the world.
- *Topological domain*: the domain where synthesis is enabled as an observable and controllable process. This domain traverses increasing levels of abstraction and is defined at a low level by the synthesis rules, syntax and semantics, and at higher levels by goal and plan generating programs and related programs for measuring progress.

Program synthesis operates on *symbolic data* - the programs that constitute the machine. It follows that such constituents must be described to allow reasoning (symbolic computation) about what they do (e.g. actions on the world), what their impact will be (prediction) - or could be, given hypothetical inputs (simulation) - what they require to run (e.g. CPU power, memory, time, pre-conditions in the world), when their execution is appropriate, etc. Such descriptions constitute *models* of the programs of the machine: models encode the machine's operational semantics. The same idea can easily be extended to entities or phenomena in the world, models then encode either (1) their operational semantics in the world through the descriptions of their apparent behaviors or, (2) the operational semantics of the machine as an entity *situated* in the world (constraints and possible actions in the world, reactions from entities, etc.).

Under operational closure the utility function is defined recursively as the set of the system's behaviors, some among the latter rewriting the former in sequential steps. But this alone does not define the *purpose* of the utility function, as mere survival is not operational in the context of machines: death is no threat to a computer, whereas failure to define and fulfill its mission shall be. To remain within the scope of this paper, suffice it to say that teleology has also to be mapped from biology onto an application domain (e.g. surviving → succeeding at discovering interesting facts on a foreign planet). Program synthesis is a process that has to be designed with regards to (meta)goals in light of the current situation and available resources. Accordingly, we see "evolution" – not natural evolution but *system evolution* – as a controlled and planned reflective process. It is essentially a *global and never-terminating process of architectural synthesis*, whose output bears at every step the semantics of instructions to perform the next rewriting step. This instantiates in a computational substrate a fundamental property of (natural) evolutionary systems called *semantic closure* (see [5, 9]). Semantic closure is [8]

> "a self-referential relation between the physical and symbolic aspects of material organizations with open-ended evolutionary potential: only material organizations capable of performing autonomous classifications in a self-referential closure, where matter assumes both physical and symbolic attributes, can maintain the functional value required for evolution".

A computational autonomous system is a dynamic agency of programs, states, goals and models and as such these assume the "physical" - i.e. constitutive - attributes mentioned above. System evolution must be observable and controllable, and thus has to be based on and driven by models, a requirement for systems engineering (coming from control theory). Some models describe the current structure and operation of the system, some others describe the synthesis steps capable of achieving goals according to internal drives, and finally yet some other models define procedures for measuring progress. To summarize, a computational structurally autonomous system is (1) situated, (2) performing in real-time, (3) based on and driven by models and, (4) operationally and semantically closed. The operational closure is a continuous program/model/goal synthesis process, and the semantic closure a continuous process of observation and control of the synthesis, that results itself from the synthesis process.

Self-Programming

We call *self-programming* the global process that animates computational structurally autonomous systems, i.e. the implementation of both the operational and semantic closures. Accordingly, a self-programming machine – the *self* - is constituted in the main by three categories of code:

- C_1: the programs that act on the world and the self (sensors[1] and actuators). These are programs that evaluate the structure and execution of code (processes) and, respectively, synthesize code; they operate in any of the three categories.
- C_2: the models that describe the programs in C_1, entities and phenomena in the world - including the self in the world - and programs in the self. Goals contextualize models and they also belong to C_2.
- C_3: the states of the self and of the world - past, present and anticipated - including the inputs/outputs of the machine.

In the absence of principles for spontaneous genesis we have to assume the existence of a set of initial hand-crafted knowledge - the bootstrap segment. It consists of ontologies, states, models, internal drives, exemplary behaviors and programming skills.

Self-programming requires a new class of programming language featuring low complexity, high expressivity and runtime reflectivity. Using any of the mainstream languages available today to write a program that generates another one and integrates it in an existing system is a real challenge. First, difficulties lie in these languages lacking explicit operational semantics: to infer the purpose of source code, a program would have to evaluate the assembly code against a formal model of the machine (hardware, operating system, libraries, etc) – the latter being definitely unavailable. Second, the language structures are not reflected at assembly level either and it is practically impossible from the sole reading of the memory to rebuild objects, functions, classes and templates: one would need a complete SysML blueprint from the designer. In other words, what is good for a human programmer is not so good for a system having to synthesize its own code in real-time. As several recent works now clearly indicate (e.g. [10, 15]), a good approach is to reduce the apparent complexity of the computational substrate (language and executive) and to code short programs in assembly-style while retaining significant

[1] Sensing is acting, i.e. building - or reusing - observation procedures to sample phenomena in selected regions of the world or the self.

expressivity. More over, self-programming is a process that reasons not only about the structure of programs but also about their execution. For example a reasoning set of programs has to be aware of the resource expenditure and time horizon of a given process, of the author (program) and conditions (input and context) of code synthesis, and of the success or failure of code invocation. The programming language must then be supported by an executive able to generate runtime data on the fly to reflect the status of program rewriting.

As a foundation to implement autonomous systems for real-world conditions, automatic theorem proving is most likely not as appropriate as it may seem in theory. Theories of universal problem solving impose actually a stringent constraint: they require the exhaustive axiomatization of the problem domain and space. For proof-based self-rewriting systems (cf. [11]) this means that *complete* axiomatization is also required for the machine itself. However, modern hardware and operating systems present such a great complexity and diversity that axiomatizing these systems is already a daunting task way out of reach of today's formal engineering methods – not to mention the practicalities of cost. More over, the pace of evolution of these components is now so fast that we would need universal standards to anchor the development of industrial systems in theory. Standards taking at least two decades from inception to wide establishment, it seems that by and large, the need for exhaustive axiomatization drives theorem proving away from industrial practice. We have no choice but to accept that theories - and knowledge in general - can only be given or constructed in partial ways, and to trade provable optimality for tractability. Self-programming has thus to be performed in an experimental way instead of a theoretical way: an autonomous system would attempt to model its constituents and update these models from experience. For example, by learning the regular regime of operation of its sensors such a system could attempt to detect malfunctions or defects. It would then adapt to this new constraint, in the fashion it adapts to changes in its environment. From his perspective, the models that specify and control adaptation (program construction) are a-priori neither general nor optimal. They operate only in specific contexts, and these are modeled only partially as the dimensions of the problem space have to be incrementally discovered and validated - or defeated - by experience, for example under the control of programs that reason defeasibly (see e.g. [7]). A system continuously modeling its own operation has to do so at multiple levels of abstraction, from the program rewriting up to the level of global processes (e.g. the utility function), thus turning eventually into a fully self-modeling system (see e.g. [4]).

Open-ended evolution requires the constant observation and discovery of phenomena: these are either external to the system (e.g. a tentacle waving) or internal - in which case they constitute the phenomenology of the self-programming process. Modeling is the identification of processes underlying this phenomenology down to the level of executable knowledge - programs. On the one hand, when no explanation is available, for example a sound featuring a distinct pitch for some reason not yet known, there is at least a *geometric saliency* we would like to capture in relevant spatio-temporal spaces. When on the other hand a phenomenon results from known dynamics, i.e. programs rewriting each other, we speak of *computational saliency*, to be observed in the system's state space. Phenomena are salient forms manifesting an underlying and possibly hidden process. They must be captured potentially at any scale - e.g. from the scale of optimizing some low-level programs to the scale of reconfiguring the entire system. Accordingly, we define states as *global and stable regimes of operation*: at the atomic level states are the stable existence of particular programs and objects (models, inputs/outputs, etc.), while higher-level states are abstract processes whose coordinates in the state space identify and/or control the execution of the programs that produce theses states. From this perspective, making sense is identifying - or provoking - causal production relationships between processes: a phenomenon P *makes sense* through the development of its pragmatics - the effects it provokes - in the system, and *P* means another phenomenon *P'* if observing *P* leads to the same (or analogue) pragmatics as for *P'*. *Making sense* is a process performed regardless of the length or duration of production graphs; it is a process that can be observed or fostered at any arbitrary scale.

Ikon Flux: an Architecture for Self-Programming Systems

Ikon Flux is a fully implemented prototypical architecture for self-programming systems - a prototype being an abstract type to be instantiated in a concrete domain. It is not the architecture of a particular autonomous system but rather a computational substrate to frame the engineering of such architectures. It is out of the scope of this paper to provide a full and detailed description of Ikon Flux (for further details see [5]); here we will focus on how this proto-architecture meets the key requirements for self-programming discussed in the previous section.

A New Computational Substrate

Ikon Flux consists of a *language* and an *executive* designed to simplify the task of programs rewriting other programs, in a unified memory distributed over a cluster of computing nodes. The language is an interpreted, functional and data-driven language. Axiomatic objects have low-level semantics (primitive types, operators and functions) and programs are stateless and have no side effects. Programs are also kept simple by virtue of abstraction: memory allocation, parallelism, code distribution, load balancing and object synchronization are all implicit. Since programs are potential inputs/outputs for other programs, they are considered data and unified as *objects*. Primitive types define prototypical and *executable* models of code structures, in the form of graphs of short (64 bytes) code fragments. Types are dynamic and

expressed in terms of other structures, which at some point derive from axioms. For example the type *program* embeds sub-structures such as a pattern (input) and code synthesis functions (outputs): these are explicit - they specify in a program-readable way the instructions to be performed by the executive and their effects - and there is no need for an additional model to describe a program's operational semantics. Ikon Flux neither defines nor allows any opaque, coarse-grained axiomatic constructs like for example long-term memory, planner or attention mechanism. High-level structures such as these have to be either hand-coded in terms of existing structures or result from code production by existing structures. However, to encompass wider technical domains (e.g. algebra, differential topology, etc.), Ikon Flux allows the manual extension of the set of primitives with user-defined code.

Programs in Ikon Flux all run in parallel, and they *react* automatically to the presence of any other object. Reaction is constrained by patterns on code structures, and *pattern matching* is the only mechanism for evaluating formal structures. Pattern matching is *deep*, i.e. patterns are, as any object, encoded in graphs and specify sub-structures and conditions at any depth in a target structure. Pattern matching is performed by the Ikon Flux executive *system-wide*, that is, (1) on any object regardless of its distribution in the cluster and, (2) patterns can be defined as combinations of multiple and inter-dependent patterns targeting different objects amongst the entire system, i.e. to identify tuples of correlated objects. Objects in Ikon Flux have a limited lifespan, controlled by a *resilience* value, and can be activated/deactivated either as input data, via an *intensity* value, or as a program via an *activation* value. Rewriting is performed upon successful pattern-matching (1) by producing new code explicitly specified in programs and (2) by modifying the control values (resilience, intensity and activation) of target objects.

Programs in Ikon Flux encode indifferently production rules or equations (with the expressivity of first order logic) and the executive performs both forward chaining (rewriting) and backward chaining. The latter has been implemented as a support for planning, but the executive is not a planning system itself: it is the responsibility of the programs to define and constrain the search space. In that respect, Ikon Flux does not provide nor does it use any heuristics: these are to be generated - and applied - by the programs themselves to control the activation/intensity values of the objects in the system.

Runtime reflective data are automatically notified by the executive and injected in the system as objects encoded in the Ikon Flux language. For example, the invocation of a function triggers the injection of a process object which in turn will be referenced by a completion object in case of termination, indicating the run time, resource usage and the program responsible for the termination if any. Process objects are also used by the executive to notify a system of any rewriting that occurred in the past.

At a higher level of organization, the language allows the construction of *internal sub-systems* as groups of objects that altogether contribute to a given function of the system. In this context, function means a process of arbitrary granularity, level of detail and abstraction that transforms the system state. Internal sub-systems are intended for macro-modeling purposes to describe global functions of the system itself, as well as behaviors, roles or functions of entities in the world. Internal sub-systems can be allocated a dedicated instance of the executive to perform rewritings in isolation from the main self-programming process: they can read and write the entire memory and their code can also be read, but not written, from their exterior. This is meant as a support for the modeling/construction of large-scale systems as recursive organizations of sub-systems.

There are some functions that cannot be expressed in the Ikon Flux language, either for efficiency reasons or because their re-implementation is too costly. Such functions are, for example, low-level audio/video signal processing, device drivers, and in general functions which do not need to evolve. Typically these functions are kept in their existing implementation and run on separate machines. Their code is hidden from rewriting programs and gathered in dedicated sub-systems called *external sub-systems*, which are wrapped in dedicated interfaces to communicate with the Ikon Flux executive. An interface consists of (1) specific axiomatic objects (types and functions) and (2) a converter to translate Ikon Flux objects into binary code invocation and vice versa. External sub-systems functions constitute the system's boundary in the world and are the only functions that *do* have side effects.

Global Semantics

Ikon Flux defines a *state space dimension* as an object constituted by $\mathbb{R}$ and the specification of an arbitrary reference process. Projecting an object on a dimension means rating its contribution (either as an input data or as a program) to the rewriting graphs that contributed to the achievement (or failure) of the reference process. This contribution is expressed by either an intensity or an activation value that is subsequently used to compute the object's final control values. Some dimensions are known a-priori, they are application-dependent and must be given by the programmer, but some are a-priori unknown, as they relate to newly discovered phenomena, and as any other object, dimensions are in general also the subject of dynamic production and decay. To allow self-reference, any object in a given system can be projected on any dimension and such a projection is an object as well. This also holds for the dimensions and sub-spaces can thus be represented symbolically in the global state space itself.

As discussed earlier, self-programming has to perform in light of goal achievement[2]. A general definition for "goal" is "a set of regions in the state space", and this implies the

[2] Geometric saliency detection is given goal semantics: the expectation of a stable form in space using a particular observation procedure – e.g. a program to detect occurrences of uncommon pitch patterns in sounds.

existence of a distance function over the state space to assess goal achievement. Thus, in addition to space dimensions, an autonomous system in Ikon Flux has to maintain and evolve distance functions, the combination of the two forming a *topological space*. As for the dimensions, the distance function is in general impossible to provide a-priori, except for the known dimensions: in general, the distance function has to be computed by the system itself, deriving programs from the programmer-supplied stock. There is, however, a particular case where the distance function can be given axiomatically: the case of *pregnance satisfaction*. Thom [12, 13] defines a *pregnance* - like hunger, fear, and reproduction in the animal reign - as an internal and global dynamics, we say *global process*, that targets abstract forms (e.g. anything edible will do for a famished dog) and constitute the ultimate reflective standard to measure the system's utility (for the dog, survival). Goals (and constraints) can be considered as *instantiated* pregnances (e.g. eating a particular bone) and they are identified thereafter as pregnances. In Ikon Flux a pregnance is implemented as an object type for specifying abstract regions in the state space, using a pattern. For example, hunger could be encoded as (1) a pregnance object P defining a pattern like "a state such as the intensity of P is lower than it is now" and (2) a program P' that modifies the intensity of P according to an underlying biological model - the execution of P' being the expression of hunger as a process. As processes, pregnances are used to define dimensions of the state space and along these, the distance function is defined by the executive: for a given object O it is the length (in time) of the shortest rewriting path - if any - that from O leads to an increase (or decrease) of the intensity of the pregnance. This measurement is computed by the executive for each object upon request by any program, e.g. when the intensity of a pregnance changes.

In Thom's Semiophysics [13], when a form, e.g. a discernable event over a noisy background, becomes salient enough under the empire of a pregnance, it can, under some conditions, trigger the contraction of event-reaction loops in shorter ones. This is called the "investing of forms by a pregnance", or *pregnance channeling*. Thom gives an example of thereof in his interpretation of Pavlov's famous experiment: the bell becomes invested by the pregnance hunger, to the point where its sole ringing triggers a response normally associated to the subsequent occurrence of the meat: the bell assumes the pragmatics of the meat. Subsumed by a pregnance, a form progressively stands for - *means* - another form. In Ikon Flux pregnance channeling can be typically implemented as the combination of (1) underlying models for pregnances - the consumption / digestion process, to which salivating is a positive contribution - (2) programs that learn an interaction pattern - occurrence of the bell, then of the meat, then of the rewriting of the event "meat" into the salivation reaction - and (3) programs that promote this pattern to a program - rewriting "bell ringing" into the invocation of the function "salivate". *Learning* is, in our example, identifying recurrent values for the projections of the events (occurrences of bell, meat and rewriting events) along the dimension associated to the pregnance hunger and related intermediate processes; feedback (or reinforcement) comes as the satisfaction of the pregnance. Pregnance channeling constitutes a semantic association between events and reactions at a global scale: as a global learning mechanism it can lead to behavior conditioning (as in Pavlov's experiment), but it is also one possible mechanism for implementing associative memories where pregnance channeling triggers the spreading of intensity control values across the system. Put briefly, plunging processes into topological spaces enables the identification of causal and dynamic relations throughout the entire state time-space for observing and controlling the system *as a whole*. This has also been demonstrated with a massively multi-agent architecture [1] whose essential parameters are represented in a topological space and controlled in real-time using morphological analysis.

Implementation

The development of Ikon Flux started in 1998, and in 2005 version 1.6 was used to test a system (Loki) in the field. The context was live theatrical performances[3] where Loki was in charge of generating and enacting the stage control according to the development of the drama. Loki was part of the full production period, adjusting its reactions to the rehearsal events like any of the actors and stage hands, given the input of the director. Loki thus constituted a fully valid crew member as human actors responded to Loki's actions in an adaptive way, and vice versa along the lines of constructivist theatre. Loki was able to control reliably and in real-time a complex machinery (wide variety of sensors/actuators and constraints) in a rich, noisy and unstructured environment under frequent and significant changes (e.g. human behaviors, live modifications of the script). It performed both in supervised and unsupervised modes (resp. during rehearsals and live shows) and for 20 days (cummulative time). Loki performed under considerable financial and time pressure and under these conditions no formal evaluation could be conducted: this is left for the development of future systems. Although this field-testing represents a limited evaluation of the full set of Ikon Flux's ideas, it verified three key features: (1) Ikon Flux theory is implementable; (2) systems implemented on its foundation can adapt in practical real-time, both on short and long timescales; and (3) Ikon Flux architectures can be reliable enough to maintain their place in a multi-person, multi-week interaction involving continuous change while meeting real goals.

So far, our experience with Loki justifies the pursuit of further developments with great confidence. Nevertheless, much work remains to be done to address all the issues pertaining to the engineering of fully autonomous systems

[3] In particular, the play Roma Amor – director J.M. Musial - premiered at the Cite des Sciences et de L'Industrie, Paris. Work supported by grants from the French Agency for Research (ANVAR) and Ministry of Culture.

like the Mars rover we described in the Introduction; there exist today no actual methodology for the *principled* design of non-trivial evolutionary systems. This is an open issue we are currently investigating on the basis of several techniques and concepts we have experimented for the construction of Loki. Here is a brief overview of these. First, in a traditional top-down fashion, we leverage prior knowledge by hand-crafting analogy-making programs. These are used to infer goals and plans by generating new heuristics from the ones provided in the bootstrap segment, and for unifying local models into more general ones. Second - and this is a more decisive research avenue - we identify and foster programmatically the formation and the transformation of high-level structures, functions and processes from the bottom up. This is required essentially (1) to measure and control the stability of functions, (2) to understand their formation for building new ones for arbitrary purposes, (3) to form concepts operationally for keeping the system complexity at a reasonable level as it produces new knowledge, and (4) to optimize existing rewrite graphs. To this end we have included in the bootstrap segment some "architectural" code: it consists of ad-hoc programs and models designed to identify and construct high-level organizations such as *functions* - stable coupling over time of inputs and effects - *organons* - ephemeral aggregates of code forming a substrate for a function, i.e. abstract patterns of computation flux - *organisms* - aggregates of organons operationally closed for a set of functions - and *individuals* - aggregates of functions, organons and organisms semantically closed for a pregnance. However, to do so in a general and deterministic way remains unsolved. Notice also that, at this stage of development, a significant part of the bootstrap segment has been kept away from evolution, notably architectural code and internal drives.

Performing deep pattern matching over a massive amount of fine-grained objects[4] in real-time is computationally intensive, but not intractable. Ikon Flux has been implemented on clusters of standard PCs running RTAI[5]. Our measurements of Loki's performance show that, under constant load, rewriting speed scales well with the number of processors; but they also show that scalability - the admissible load - is severely limited by current networks and protocols (TCP/IP over Ethernet). As a result, a new version (2.0) is planned, targeted at multi-core processors communicating by RDMA over Infiniband.

Conclusion

In the domain of computational systems, autonomy can be operationalized by the concept of self-programming for which we have presented a prototype: a dynamic, real-time and self-referential architecture for building and grounding knowledge and processes in high-dimensional topological spaces. For such systems, harnessing the emergence of high-order organizations in a general scalable way calls for new engineering methodologies. As we have argued here, these must be based on a process-oriented and generative approach. To this end we are currently investigating the possibility of modeling self-programming using Goldfarb's Evolving Transformation System [3].

[4] Loki was constituted by roughly 300 000 objects in average.

[5] A real-time kernel extension for Linux [www.rtai.org] paired with RTnet a real-time network protocol stack [www.rts.uni-hannover.de/rtnet]

References

[1] Campagne J.C. Morphologie et systèmes multi-agents. Ph.D. thesis, Université Pierre et Marie Curie, Paris. 2005

[2] Froese T., Virgo N., Izquierdo E. Autonomy: A review and a reappraisal. In F. Almeida e Costa et al. eds. *Proc. of the 9th European Conference on Artificial Life*. Springer-Verlag, Berlin. 2007

[3] Goldfarb L., Gay D. What is a structural representation? Fifth variation, Faculty of Computer Science, University of New Brunswick, Technical Report TR05-175. 2005

[4] Landauer C., Bellman K.L. Self-Modeling Systems. In R. Laddaga, H. Shrobe eds. *Self-Adaptive Software: Applications*. *Springer Lecture Notes in Computer Science*, 2614:238-256. 2003

[5] Nivel E. Ikon Flux 2.0. Reykjavik University, School of Computer Science Technical Report. RUTR-CS07006. 2007

[6] Pattee H. *Evolving Self-Reference: Matter, Symbols, and Semantic Closure*. In *Communication and Cognition*. *Artificial Intelligence*, 12(1-2). 1995

[7] Pollock J. OSCAR: An agent architecture based on defeasible reasoning. In *Proc. of the 2008 AAAI Spring Symposium on Architectures for Intelligent Theory-Based Agents*. 2008

[8] Rocha L.M. ed. *Communication and Cognition. Artificial Intelligence*, Vol. 12, Nos. 1-2, pp. 3-8, Special Issue *Self-Reference in Biological and Cognitive Systems*. 1995

[9] Rocha L.M. Syntactic autonomy, cellular automata, and RNA editing: or why self-organization needs symbols to evolve and how it might evolve them. In Chandler J.L.R. and G, Van de Vijver eds. *Closure: Emergent Organizations and Their Dynamics*. *Annals of the New York Academy of Sciences*, 901:207-223. 2000

[10] Schmidhuber J. Optimal Ordered Problem Solver. *Machine Learning,* 54, 211-254. 2004

[11] Schmidhuber J. Gödel machines: Fully Self-Referential Optimal Universal Self-Improvers. In Goertzel B. and Pennachin C. eds. *Artificial General Intelligence,* p. 119-226, 2006.

[12] Thom R. *Structural Stability and Morphogenesis*. Reading, MA: W. A. Benjamin. 1972

[13] Thom R. *Semiophysics: A Sketch*. Redwood City: Addison-Wesley. 1990

[14] Varela F.J., Maturana H.R. *Autopoiesis and Cognition: The Realization of the Living*. Boston, MA: Reidel. 1980

[15] Yamamoto L., Schreckling D., Meyer T. Self-Replicating and Self-Modifying Programs in Fraglets. *Proc. of the 2nd International Conference on Bio-Inspired Models of Network, Information, and Computing Systems*. 2007