HÁSKÓLINN Í REYKJAVÍK
REYKJAVÍK UNIVERSITY

Ikon Flux 2.0

Eric Nivel

RUTR - CS07006

Dec. 2007

Reykjavik University

Department of Computer Science

Technical Report

Abstract. Structurally autonomous systems embody a continuous process of adaptation of their internal structure to new situations, goals and constraints in open-ended environments and in real-time. An operational definition for autonomy is proposed: self-programming, that is, a continuous process of architectural synthesis. Ikon Flux is described, a proto-architecture consisting of a language and an executive, designed to enable the construction of autonomous systems as self-programming systems. The prototype of such a system is a dynamic and self-referential architecture generating in real-time its own structures and processes, both grounded mutually in high-dimensional topological spaces. A brief comparison of Ikon Flux with prominent state-of-the-art architectures is given. I discuss some salient issues pertaining to engineering evolutionary systems and conclude to the necessity of designing a new methodology based on process generativity.

1 Introduction

The work presented here aims at the construction of machines able to adapt to unforeseen situations in open-ended environments. “Adaptivity” 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 knowledge and resources - including time.

Ikon Flux is a prototypical architecture for such autonomous systems. It is not *the* architecture of a particular system per se, but an executive and a language to frame the engineering of such architectures in the sense that a prototype is an abstract type, to be instantiated in a concrete domain.

Ikon Flux has been implemented: its development started in 1998, and version 1.6 was finalized in 2005 when an actual system – Loki - was built to act as a meta character in theatrical performances[1] and in particular, the play Roma Amor[2] premiered at the Cite des Sciences et de L’Industrie in Paris. Its function was to generate and enact the stage control according to the development of the drama, *a la manière de* characters in the play, present on stage or not. In return, human characters responded to the system in an adaptive way: the system formed by the script, the actors and the machinery then assumed some degree of autonomy, along the lines of the constructivist approach to theatre[3].

This paper describes the version 2.0 of Ikon Flux currently under redevelopment. Ikon Flux V2.0 uses the same fundamental concepts as version 1.6 and is intended to bring improvements on two main fronts: the performance of the executive and the syntax of the programming language.

This report presents three levels of reading:

- Theoretical level - section 2: an operational definition of autonomy as a self-programming process,
- Operational level – section 3: a description of the core concepts of Ikon Flux, that support the construction of self-programming systems,
- Technical level – section 4: a description of the Ikon Flux executive. Also, the Annex provides a definition of the Ikon Flux language: code synthesis rules, built-in operators, object types and system functions, and also some code examples.

Finally, section 5 provides a brief discussion about related architectures.

2 Self-Programming: Operationalizing Autonomy

Lacking an operational definition for the concept of autonomy has considerably weakened its impact on system engineering. In practice, so-called “autonomous” systems are hardly autonomous at all since they are primarily built only to address situations that can be described beforehand. The mainstream approach towards system engineering today consists essentially of building sophisticated ways to select and tune hard-coded goals and behaviors for handling situations framed in hard-coded ontologies. Systems designed this way belong to the category of *behaviorally* autonomous systems (Froese et al. 2007) - systems that, essentially, embody task automation. The definition of both their tasks and their domain of operation is pre-determined and, by and large, such systems are not aware of their own purpose: they are defined entirely by a set of target values and mechanisms to reach them, both imposed from outside their reach. Shall these values and mechanisms become obsolete in a changing world, such systems will cease to operate correctly, if at all. In other words, they are ignorant of the the laws of both their construction and their determination, and

[1] Work supported by grants from the French Agency for Research (ANVAR) and the Ministry of Culture.

[2] J.M. Musial, director.

[3] See for example Meyerhold’s acting system – Biomechanics. See also Meyerhold (1969) and Hoover (1974). For a more general perspective, see http://www.univie.ac.at/constructivism.

sensing the world is for them to recognize events using the sieve of an immutable phenomenology. Defined from the outside in, these systems cannot – by design – adapt to change without the intervention of their programmers: in fact, such systems fall directly in the category of *allonomic* systems (systems governed exclusively by external laws), the exact opposite of what autonomous systems are (governed entirely by their *own* law)!

If the goal of building autonomous systems is to be pursued in an authentic way, then motivations, goals and behaviors shall be envisioned as being dynamically (re)constructed by the machine as a result of changes in its internal structure, the latter resulting from changes in the environment, which in turn is modified by the system's behaviors. This perspective - *structural coupling* - draws on Varela's conceptual framework, namely *autopoiesis*, and his work on *operationally closed* systems (Varela et al. 1974, Varela and Maturana 1980, Varela 1992) that is:

> *"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 autopoiesis is a model designed primarily to describe the formation of complex organizations in bio-chemical substrates, it constitutes an abstract theory as it does not impose any constraint on the substrate. Autopoiesis defines relations between the components, processes, network and topological domains rather than it describes them individually. In that sense autopoiesis has to be made operational in a particular domain, that is, to be instantiated by the specification of the entities mentioned above. For example, Fontana (1996, see also Ancel and Fontana 2000), has proposed a computable model for chemistry and RNA folding based on lambda calculus where components (molecules) and networks (chemical reactions) are specified as lambda-terms and reduction rules. Addressing the case of computational substrates I 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.
- *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 progress measuring programs.

Program synthesis operates on *symbolic data*, i.e. 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: mere survival is not operational in the context of machines - death is no threat to a computer, whereas failure to (re)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. in the case of

an exploration robot: 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, system evolution shall be seen 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 Pattee 1995 and Rocha 2000). Semantic closure is (Rocha 1995)

> *"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 system engineering - see for example Sanz et al. (2007). 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 differentiate the approach described here form the misleadingly named "behavioral autonomy" I have – hopefully temporarily – to resort to using the term "structural autonomy". To summarize,

a computational structurally autonomous system is a system

- *situated,*
- *performing in real-time,*
- *based on and driven by models,*
- *operationally and semantically closed.*

The operational closure is a continuous program/model synthesis process, and the semantic closure a continuous process of observation and control of the synthesis, that results itself from the synthesis process.

2.1 Self-Programming

Self-programming is 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[4] and actuators). Sensors and actuators on the self operate in any of the three categories: they are programs that evaluate the structure and execution of code (processes) and, respectively, synthesize code.
- C_2: the models that describe 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.

A new class of programming languages. Self-programming requires a new class of programming language featuring low complexity, high expressivity and runtime reflectivity. As a case in point, C++ is one of the most powerful languages today for engineering real-world systems: it offers both performance, direct access to computing resources and high-level abstractions. But for a C++ program to generate another one is a real challenge: difficulties lie for example in its syntax being irregular and too permissive, and – more dramatically - in its operational semantics being not explicit, i.e. not expressed as

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

C++ objects. The object-orientation paradigm has been introduced for project management purposes and to instrument a static top-down analysis that *excludes* bottom-up dynamic code production. Operational semantics being unavailable, a program would have to infer the purpose of source code by evaluating the assembly code against a formal model of the machine (hardware, operating system, libraries, etc) – the latter being definitely unavailable for the off-the-shelf components we use daily. Besides, 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 so far – and even that is disputable! - for a human programmer is unsuitable for a system having to synthesize its own code in real-time. This discussion also holds for civilized languages like Erlang or Haskell, for which the principal difficulties still prevail – lack of operational semantics and exclusive top-down orientation.

A good approach towards addressing this issue is to reduce the apparent complexity of the computational substrate (language and executive) and to code short programs in assembly-style while retaining significant expressivity. There seem now to be a consensus on this approach, as several recent works indicate (for example, Paun 2002, Schmidhuber 2004, Spector et al. 2005, Yamamoto et al. 2007).

In addition, runtime data must be generated on the fly to reflect the status of program rewriting for self-programming is a process that reasons not only about the structure of programs but also about their execution (processes). For example a reasoning (set of) program(s) has to be aware of (1) the resource expenditure and time horizon of a given process, (2) the author (program) and conditions (input and context) of code synthesis, and (3) the success or failure of code invocation.

An experimental approach to system modeling. As a foundation for implementing autonomous systems in and 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 – see for example Schmidhuber (2006) - this means that *complete* axiomatization is also required for the machine itself. However this is very unlikely to happen before long: modern hardware and operating systems present such complexity and diversity that axiomatization is already a daunting task way out of reach of today's formal engineering methods – not even mentioning the cost issue. 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. Bearing in mind that standards take in general at least a decade to reach maturity and an extra one to be widely established, 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 perform 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 detect malfunctions or defects. It would then adapt to this new constraints, 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 Pollock 2001). A system continuously modeling its own operation has to do so at increasing level of abstraction, from the level of program rewriting up to the level of global processes (e.g. the utility function), thus turning eventually into a fully self-modeling system - see Landauer and Bellman (2003).

Multi-scale global semantics. In that respect, self-programming systems need to capture and construct operational knowledge at any arbitrary - and *global* - scale. Open-ended evolution requires the constant observation and discovery of phenomena: these are either external to the system (they occur in the world) or internal – in which case they constitute the phenomenology of the self-programming process. In that respect, 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 known yet, 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, I speak of *computational saliency* to be observed in the system's state space. Phenomena are salient forms manifesting an underlying and potentially hidden process. They must be captured possibly at any scale – e.g. from the scale of optimizing some low-level programs to the scale of reconfiguring the entire system.

Accordingly, I define states as *global and stable regimes of operation* – intuitively, stable phenomena and their underlying processes, either already known or still to be discovered. At the lowest level, states are the stable existence of particular (set of) programs (and objects like models, inputs/outputs, etc.), while higher level states are abstract processes whose coordinates in space control the execution of the programs that implement theses states. The dimensions of the state space are thus dynamic objects and they bear the semantics of processes – expressed in practice as rewrite graphs. From this perspective, sense making is identifying – or provoking - causal production relationships between processes: a phenomenon P makes sense through the development of its effects in the system, and P means another phenomenon P' if observing P leads to the same (or analogue) effects as for P'. Sense making is performed regardless of the length or duration of rewriting graphs: it is a process that can be observed or fostered at any arbitrary scale.

3 Core Concepts

Self-programming systems require a programming language designed for efficient and tractable dynamic code synthesis. It also requires support for capturing the operational semantics of a system at multiple and potentially global scales. This section describes how these key requirements are addressed in Ikon Flux. References to the annex are provided (in boldface) whenever concepts are mapped directly to language constructs.

3.1 A Language For Code Synthesis

The Ikon Flux language has been designed for simplifying the task of programs rewriting other programs. For the sake of scalability, rewritings are performed in a unified memory distributed over a cluster of computing nodes. The Ikon Flux language is an interpreted functional language with low-level axiomatic objects (primitive types, operators and functions) and programs written in it 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, all are implicit.

Objects. 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[5] code fragments. For example the type *program* (**pgm**) defines the specification of a pattern (**spc**) and the specification of code synthesis (modification of existing objects or production of new ones), and there is no need for an additional model to describe its operational semantics - in the case of programs: "when an object has the form defined by the pattern, the system is added the code specified in the synthesis segment". Structures and structure composition are both explicit: in general, types are dynamic and expressed in terms of other structures, which at some point derive from axioms. Ikon Flux neither defines nor allows any opaque and coarse-grained axiomatic constructs, like for example long-term memory, planner, attention mechanism, etc. The general orientation is that if high-level structures such as these are needed, then they 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 objects (see 4.2).

[5] 64 bytes for fragments, 88 for sys-objects.

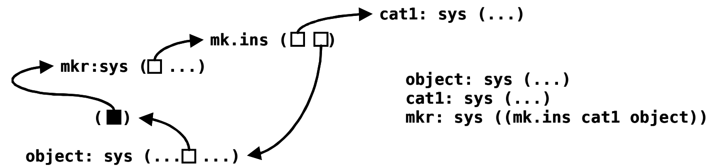


Figure 1 – Objects and fragments. *Sys-objects (**sys**) are objects that live freely in the memory, whereas fragments are the internal constituents of said sys-objects and are bound to them. Here three sys-objects (`object`, `mkr`,`cat1`) are represented. Any object can reference others (e.g. `mkr` referencing `mk.ins`) and this list of references constitutes a sub-structure of a given sys-object (e.g. black block in `object`). NB: dots (...) denote the existence of structures not represented for the sake of readability.*

Pattern matching. From a low level perspective, 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 valuating formal structures. Pattern matching is *deep*, i.e. patterns – as any object they are encoded in graphs – can specify sub-structures and conditions at any depth in a target structure.

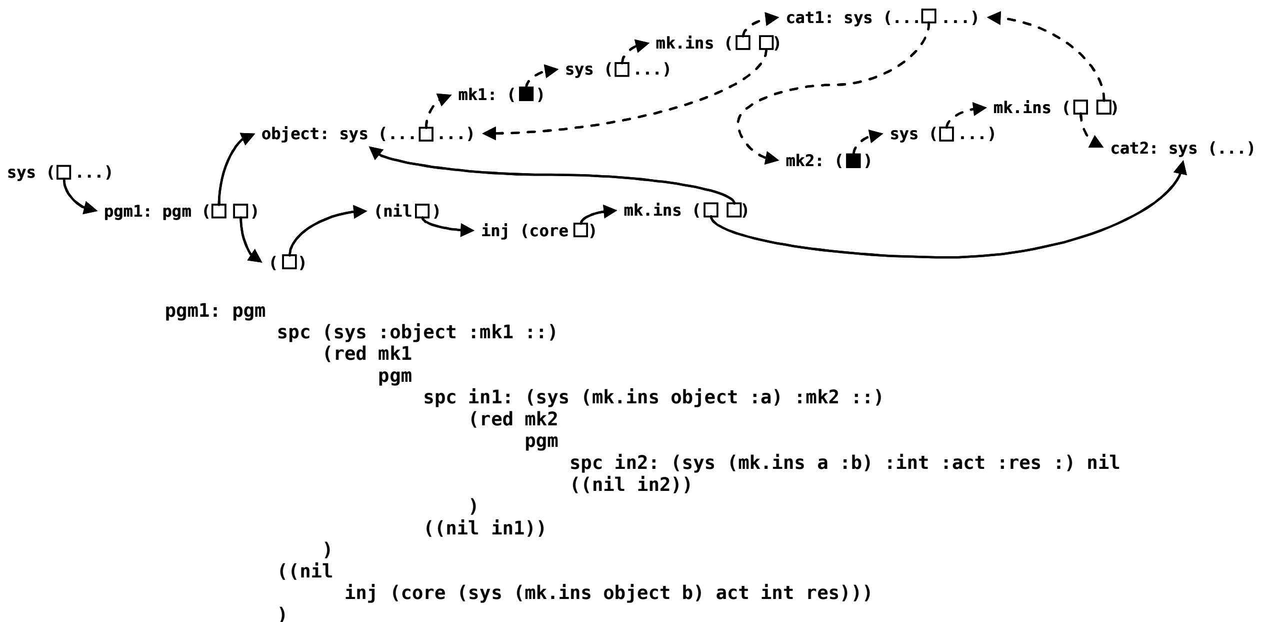


Figure 2 - Deep pattern-matching. *Pattern matching is performed by and on any objects. Patterns specify incomplete composition of sub-structures of arbitrary depths. Here a program (`pgm1`) is represented defining a pattern (`object`, dashed structure composition pointers). The program states that whenever an object is an instance (`mk.ins`) of a category object (`cat1`) and this category is itself an instance of another category (`cat2`) then a new tag is added to the system that expresses that object is also an instance of `cat2`.*

Pattern matching is performed by the Ikon Flux executive (see section XXX for a description of the executive) *system-wide*, that is, (1) on any object regardless of its location in the computer cluster, and (2) patterns can be defined as combinations of multiple and inter-dependent patterns targeting different objects amongst the entire system, i.e. for the purpose of identifying tuples of correlated objects.

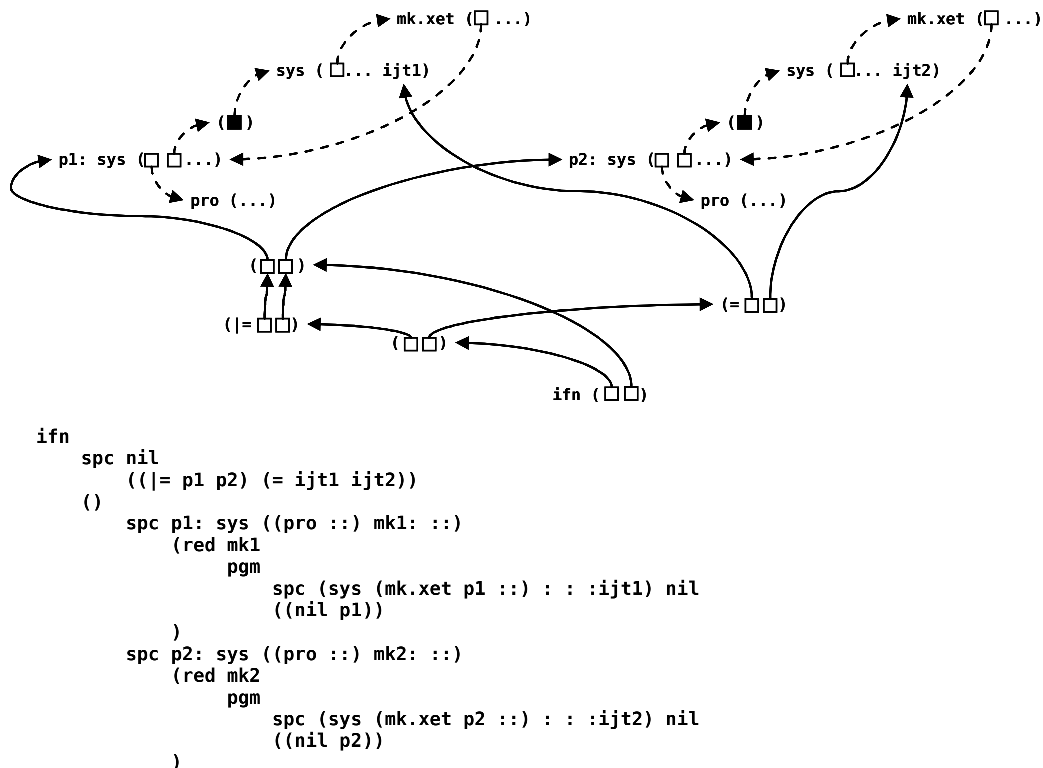


Figure 3 – System-wide pattern-matching. *Pattern matching can be performed on multiple sys-objects simultaneously. Here a pattern specification is represented targeting pairs of objects. The specification is a composition of patterns which can share fragments and/or references to any of the input objects. Here, the pattern (defined by the implicit function* `ifn` *(**efn**)) detects processes that are different (*`|=`*) and terminate at the same time (*`=` *on the injection times* `ijt` *of completion markers (**xet**)).*

Objects in Ikon Flux have a limited lifespan (controlled by a *resilience* value) and also, can be activated/deactivated either as input data (*intensity* value) or program (*activation* value). Rewriting is performed upon successful pattern-matching (1) by producing new code[6] explicitly specified in programs and (2) by modifying the control values (resilience, intensity and activation) of target objects.

Reflectivity. 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 (**pro**) which in turn will be referenced by a completion object in case of termination, indicating the run time and resource usage and also the program responsible for the termination in case of non natural death. Processes are also used by the executive to notify a system of any rewriting that occurred in the past. At a higher level, "reflectivity" means reflecting in objects the effects of the computation on the system and on the world. This is achieved by models, ontologies and observation procedures, provided by the programmer either directly – i.e. as is - or indirectly - i.e. produced dynamically by intermediate programs deriving ultimately from hand-crafted knowledge.

Forward/Backward Chaining. As in mixed-paradigm languages (e.g. functional/logical in Curry[7] or Maude[8]), programs in Ikon Flux encode indifferently production rules (**pgm**) and equational rewrite rules (with the expressivity of first order logic - **erw**) and the executive offers functions to perform both forward chaining (rewriting – **eva**, **red**) and backward chaining (**inv**, **ctx**). Backward chaining 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 programs to control the activation/intensity values of the objects in the system.

[6] NB: duplicates of code are automatically eliminated – multisets are not supported.

[7] See http://www.informatik.uni-kiel.de/~curry/

[8] See http://maude.cs.uiuc.edu/

Internal sub-systems. From a higher-level perspective, the language allows the construction of sub-systems as groups of objects that altogether contribute to a given or an emergent function of the system - a function being here a process of arbitrary granularity, level of detail and abstraction that transforms the system state in some way. Sub-systems (**ent**) are meant for macro-modeling purposes: they are used to describe functions of the system itself, behaviors, roles or functions of entities in the world, etc. Sub-systems can be allocated a dedicated instance of the executive (**spv**) 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 feature is meant as a support for (1) the modeling/construction of large-scale systems as recursive organizations of (potentially) autonomous sub-systems and (2) for designing, debugging and monitoring purposes (also called supervision): this allows for example, profiling code to run in isolation from the system under evaluation so that time readings are not including profiling run time.

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

3.2 Topological Spaces

Self-programming systems have to model dynamically their own operation. This means that every process in the system has to be accounted for in terms of its effects on the system – at the lowest level (resource usage), at the highest level (contexts and goals) and also at intermediary levels (rewritings). Ikon Flux adopts a *process-centric* approach to modeling the operational semantics of a system at multiple scales. It is essentially plunging processes into topological spaces to measure their contribution to the achievement of other processes. Each process is projected into several dimensions each representing another process like for example, the execution of a program (concrete atomic process), the observation of a salient value (abstract atomic process), a rewriting graph (concrete composite process), or the achievement of a goal (abstract composite process).

Dimensions. Ikon Flux defines a state space dimension as an object (of type *realm* - **rlm**) constituted essentially by $\mathbb{R}^*$ and the specification of an arbitrary reference process. 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. As discussed earlier, self-programming has to perform in light of goal achievement. A general definition for "goal" is "a set of regions in the state space", and this implies the existence of a distance function over the state space: only under this condition is it possible to assert whether or not the system is in a given state and therefore whether it has achieved its goals or not, not to mention the necessity of measuring progress. Thus, in addition to spatial dimensions, an autonomous system has to define a distance function, which in combination with the dimensions forms a *topological space*.

In Ikon Flux, the distance function is the measure of the contribution of an object (either as an input data or as a program) to the rewriting graphs that impacted the achievement (or failure) of the reference process. For example, if the process is implemented as a rewrite graph, then positively contributing objects are objects that intensify, activate or produce the elements of the process' graph. This contribution is encoded as the depth – along the execution time scale - of an object in the rewrite graph that contributed to the reference process[9].

The contribution of an object to the achievement of the reference process is called a projection of the object onto the dimension associated to the process. The executive projects

[9] NB: in case the object can be found at several locations in the graph, the smallest depth is retained.

every object in a system upon request by programs (calling the function **ctx** for a given realm).

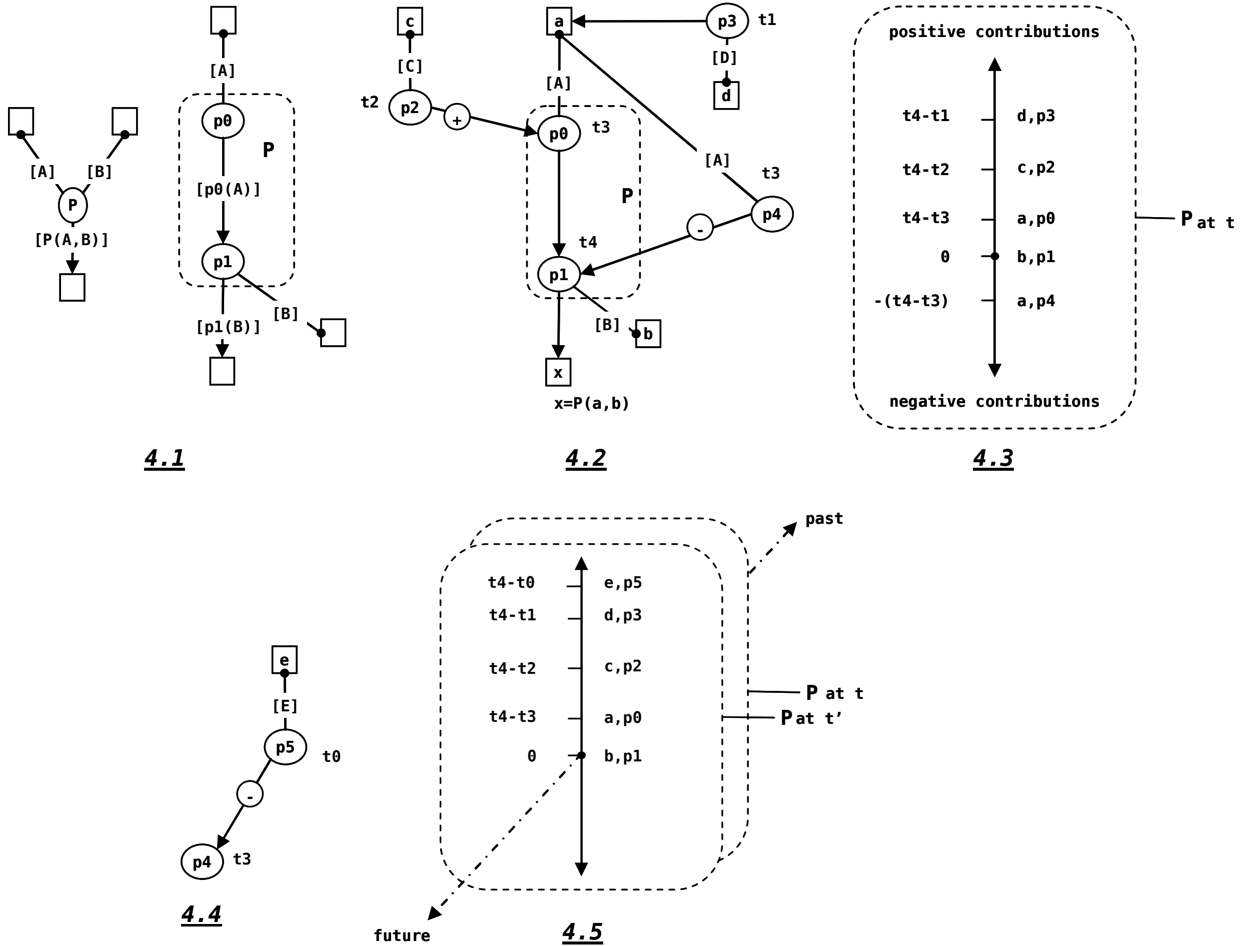


***Figure 4 - Evaluating the contribution of rewrite graphs to a process**. Fig. 4.1: let an abstract process P reacting to the patterns A and B, and producing objects patterned after P(A,B). Let P be implemented by an actual rewrite graph (dashed enclosure). Fig. 4.2: by time t=t4, several other processes have contributed positively (p2 increasing the activation value of p0, p3 producing a, an input for p0) or negatively (p4 decreasing the activation value of p1). Fig. 4.3: the performance times of these contributors are recorded on the realm associated to P. Fig. 4.4: at the time t=t' of a subsequent run of P, shall a process p5 successfully deactivate p4 (negative contributor to P), then a new instance of the realm would be created, featuring p5 as a positive contributor – NB: p4 would disappear from the realm as it could not run anymore. Notice also that contribution evaluation does not necessarily depend on a particular scale: for example p0, p1, etc. could be abstract processes as well (as is P), and in that case, the contribution of their constituents could also be rated with regards to P. More over should P exert feedback on its contributors, P would be rated in their respective realms (e.g. the realms associated to p2, p3, p4 and p5).*

Control. There exists an axiomatic dimension (the system realm, also called *sys-realm*) that does not refer to any reference process. It merely holds projections for every object in the system; such projections are directly defined as the final intensity and control values (instead of contributions to a reference process). Contribution values are translated into either intensity or activation control values depending on the role of the object in the graph (input data or program), and a control value is computed as the inverse of a contribution value. Control values are thus locally bound to a dimension but can be subsequently combined[10] as final values on the sys-realm: these will eventually be used by the executive to control a given object. Each realm defines two pairs of thresholds: one for the positive

[10] Programs are responsible for defining/selecting the method for combining projections, i.e. average, maximum, etc.

contributions, one for the negative ones, and in each pair there is a threshold for input data and another one for programs. On the sys-realm, these thresholds pairs control intensity and activation control values: only objects with a final intensity value above an intensity threshold will be considered by the executive as potential inputs for programs; symmetrically, only programs with a final activation value above the activation thresholds will be considered by the executive for execution.

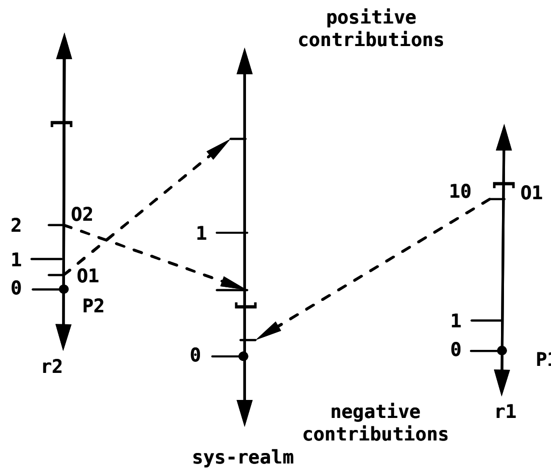
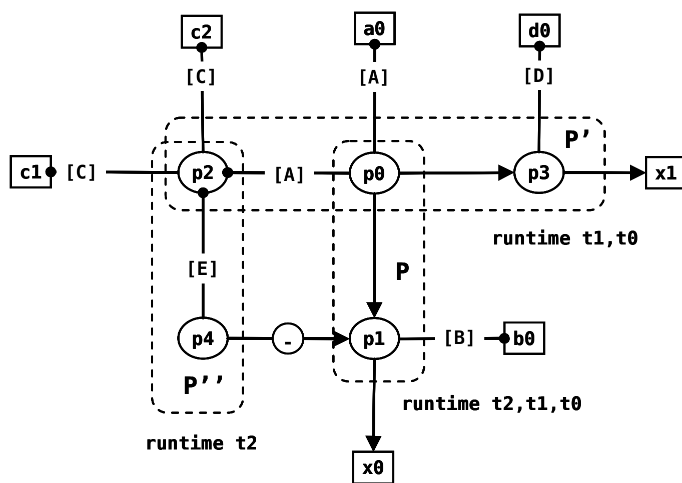


Figure 5 - Translating contributions to control values*. In general, the deeper an object in a rewrite graph the less likely it is to be finally intense or active. However, this only holds locally, i.e. for a given realm. Shall an object (like o_1) be projected on another realm, its final control value will be a combination of its local contributions - see* ***rlm*** *for the methods of combination. If the maximal local value was chosen, o_2 would be activated (activation thresholds indicated by brackets). NB: it is also allowed to define or modify control values explicitly, i.e. programmatically.*

As a support for self-reference, any object in a given system can be projected on any dimension and such a projection is an object as well (**pja**, **pji**). This holds also for dimensions themselves and sub-spaces in the state space can thus be represented symbolically in the global state space itself (the sys-realm). As for objects in general, projecting a dimension onto another one means evaluating its contribution to a given reference process. In this particular case, the contribution of a dimension D to a process P is the multiplication of the contribution of the first object O being its best contributor (and having contributed to P) with the contribution of O to P. This is calculated for the two domains of positive/negative contributions and the highest absolute value is retained, thus determining the domain of the contribution (positive or negative). Projected that way, dimensions also receive a final control value (intensity) in the sys-realm. When a dimension gets under the intensity threshold, all the projections it holds are discarded for the computation of final control values; symmetrically, when it gets back above the threshold, the related projections are taken back into account. In practice, intensifying or deintensifying realms controls the intensity or activation of objects forming entire subsets of the system, i.e. *system configurations*. System configurations can be formed either by projecting dimensions as discussed, or alternatively by explicitly (programmatically) modifying the final intensity values of such dimensions.



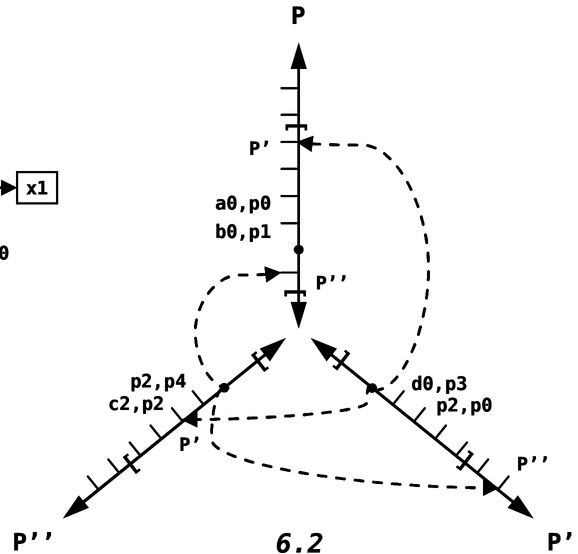
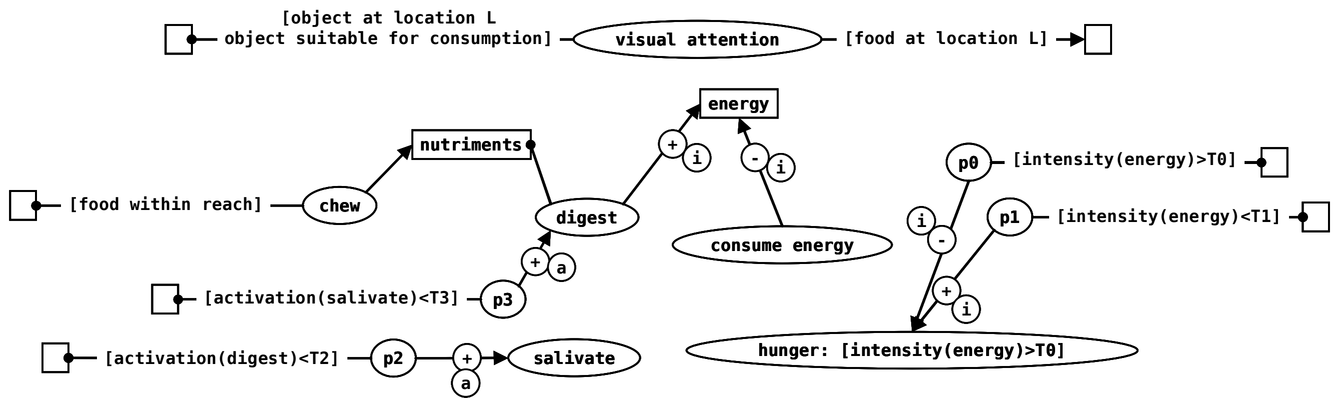


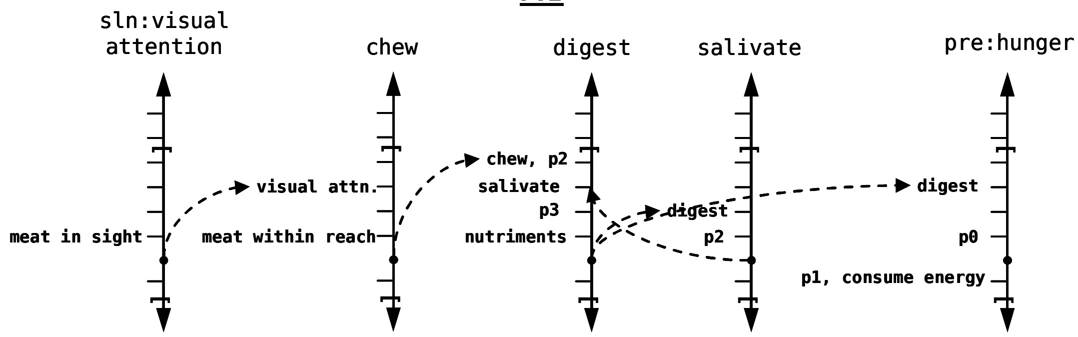
Figure 6 - System configurations. *Fig 6.1: with regards to process P, the contribution of P' is the contribution of P2 and P0. The projection of the realm associated to P' onto the realm associated to P holds the product of the contribution of P0 to P (the best contributor to P in P') and the contribution of P0 to P'. If the value of the projection of P' onto P gets above the activation thresholds defined for P, then p2 and p3 would be deactivated, but not p0 since as a main contributor to P already accounted for in the dimension associated with P its projection would be probably well below the threshold. These rules also apply to P'', and as it contributes negatively to P, its constituents would be also deactivated if the goal was to maintain P. Fig. 6.2: projections of P' on P, P'' on P, P' on P'' and P'' on P'. Pairs of activation thresholds are indicated by brackets; in this example thresholds are all set to the same values on their respective realms.*

Goal semantics. Thom (1972 and 1990) defines a *pregnance* – like hunger, fear, and reproduction in the animal reign – as an internal and global dynamics, i.e. a *global process*, that targets abstract forms (e.g. anything edible will do for a famished dog) and pregnances constitute the ultimate reflective standard to measure the system's utility (for the dog, survival). Goals can be considered as *instantiated* pregnances (e.g. eating a particular bone) and they are identified thereafter to pregnances. In Ikon Flux, a pregnance is implemented as a type (**pre**) 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 (simulated) biological model – the execution of P' being the expression of hunger as a process – see fig. 7 below. As processes, pregnances are used to define dimensions of the state space and along these, the distance function is defined along the same line as for general processes. In the particular case of a pregnance, the distance function is, for a given object O, the length – along the execution time scale - of the shortest rewriting path – if any – that from O leads to an increase (or decrease) of the intensity of the pregnance. As for general processes, this measurement is computed by the executive for any object and for one dimension upon request by any program, e.g. whenever the intensity of a pregnance changes.

For unification, geometric saliency (**sln**) detection is given goal semantics: the expectation of a stable form in space using a particular observation procedure – for example, a program detecting occurrences of uncommon pitch patterns in sounds.



7.1



7.2

Figure 7 – A (simplified) model for the pregnance hunger. *Rating the contribution of rewrite graphs in a dimension associated to a pregnance. Fig. 7.1: a simulation of a biological substrate. The salivation process stimulates the digestion process (+a) via p3, whereas p2 acts symmetrically. The pregnance hunger is controlled in intensity by p0 and p1. Fig. 7.2: the resulting projections of processes and objects on the significant realms (realms for p_i are not represented). NB: visual attention is a geometric saliency (**sln**) detection process.*

Sense making. In Thom’s semiophysics (Thom 1990), when a form (like 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 this 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 only with 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. In our computational substrate, pregnance channeling can easily be implemented as the combination of (1) programs that learn an interaction pattern – e.g. occurrence of the bell, then of the meat, then of the rewriting of the event “meat” into the salivation reaction – (2) underlying models for pregnances – e.g. the consumption / digestion process, for which salivating is a positive contribution - and (3) programs that turn this pattern into a process upon repeated pregnance satisfaction – e.g. rewriting “bell ringing” into the invocation of the function “salivate”. Learning is, in this example, identifying recurrent values for the projections of events (e.g. occurrences of bell, meat and rewriting events) along the dimensions associated to the pregnance hunger and to related intermediate processes (e.g. consumption / digestion); feedback (or reinforcement) comes as the satisfaction of the pregnance – see fig. 8 below.

Pregnance channeling constitutes a semantic association between events and reactions at a global scale. As a global learning mechanism, pregnance channeling instruments behavior conditioning (Pavlov’s experiment), and also constitutes one possible mechanism for implementing a general associative memory – pregnance channeling would trigger the spreading of intensity control values across the system. From a more general perspective, pregnance channeling is, in the main, instrumented by plunging reflective processes in topological spaces for identifying causal and dynamic relations at any scale. This approach is a general solution for observing and controlling a system *as a whole* – for a related way of achieving global control, see Campagne (2005). This solution is general for it is an abstract

and structural dynamics – it operates on rewriting semantics and does not depend on any application domain - and also because it is independent from scale.

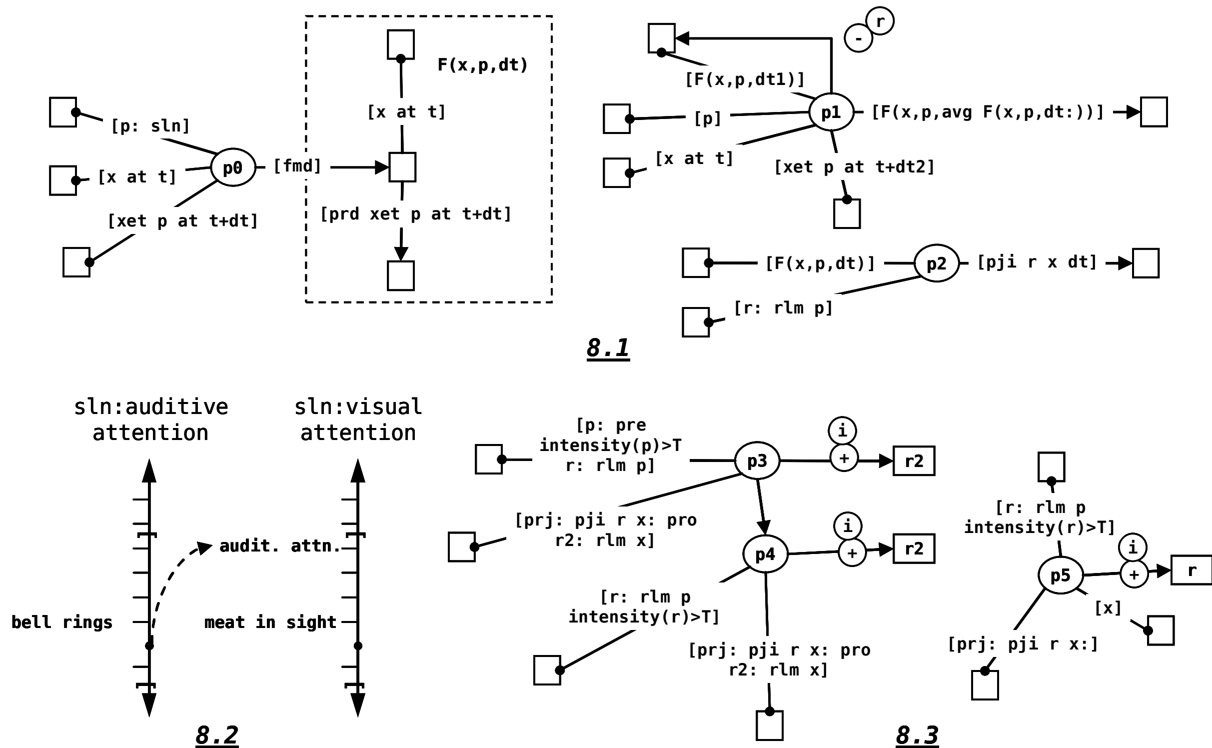


Figure 8 – A (simplified) model for pregnance channeling. *Here are represented the main processes and associated realms for implementing the example discussed above. It is an addition to the model presented in figure 7. Fig 8.1: p0 generates a forward model (**fmd**) that predicts the occurrence of an event (here the termination (**xet**) of a salient process p) given a harbinger (x). This sequence is learnt by p1 and p2. P1 creates a new forward model whenever the occurrence of x is followed by the process termination with a different delay (dt2 instead of dt1); the old model is destroyed (the resilience r is decreased) and a new one is created (here taking the average (**avg**) of the observed delays). P2 projects the harbinger onto the realm associated with the salient process. Fig 8.2: the resulting projection. Fig. 8.3: exploitation. P3 and p4 propagate intensities to realms contributing starting from the pregnance realm (backwards). P5 forwards the intensity of a projection to the realm it belongs to. In our example, when the bell rings, p5 intensifies the auditive attention realm, then the visual attention realm and so on until the pregnance realm. The control mechanism that translates contributions values from realms to actual intensity/activation values on the sys-realm will result in activating every process involved, in particular: chew, digest, salivate. Only the salivation program can act with no input, and will therefore be executed.*

Analogy making. The experimental nature of system modeling calls for experimental model construction procedures. From this perspective, modeling is essentially reasoning about actions with insufficient and uncertain knowledge. For example, available models may be limited to the description of the outcome of specific actions in particular contexts. In order to explore wider contexts, self-programming systems have to build hypotheses and validate or invalidate them by evaluating the results of experiments. Exploring wider contexts – i.e. states - is for a modeling system applying available models to different – but somehow related – states. One way to do so is producing analogies between models and/or between states.

Ikon Flux defines models as objects that encode either (1) the outcome of an action (*forward models* - **fmd**) or (2) the actions to be performed to achieve an arbitrary outcome (*inverse models* - **imd**) – for further details on forward/inverse models, see Wolpert and Kawato (1998). "Outcomes" are objects that encode the probability of a state transition, and model analogy is defined as follows:

- Let a forward model M_1 encoding the state transition ST_1 resulting from the performance of an action A_1 from state S.
- Let a forward model M_2 encoding the state transition ST_2 resulting from the performance of an action A_2 from state S.

- M_1 and M_2 are analogues when there exist (1) a process P and (2) a set of forward models M^i that specifies that ST_1 and ST_2 have the same meaning for P. “Same meaning” means technically that each M^i would encode that substituting ST_2 to ST_1 in a given rewrite graph R^i already represented along the dimension of P would yield another graph featuring the same contributions to P. NB: actions A_1 and A_2 – and respectively, models M_1 and M_2 - are said to be analogues with respect to S, P and $\{M^i\}$.

Making an analogy in Ikon Flux is building models such as M^i, and using an analogy is to build a program that from any occurrence of the action A_1 in the state S produces ST_2 as an outcome – and vice and versa.

This definition can easily be extended to inverse models, briefly: two inverse models M_1 and M_2 are said analogue if there exist a process P and a set of forward models M^i encoding that performing the actions specified by either M_1 or M_2 have outcomes that have the same meaning for P. In this context, using an analogy is building a program that from any program expressing that a pregnance can be satisfied by invoking M_1, produces another program that expresses that the same result can be achieved using M_2.

Analogies can also be performed on states as well, and here again would be encoded in models: a set of forward models can express that the states in question are the outcome of analogue actions, while a set of inverse models would express that to reach these states, analogues actions can be performed.

Performing analogies makes use of contextual knowledge: models, states and processes. These contexts in turn can also be defined by analogy: for example by substituting the term “same” with the term “analogue” in the definition of analogy between forward models. Notice that analogy operations – however sophisticated and contextualized – are, in the end, all anchored in axioms, i.e. equality check and pattern matching. For example, performing analogies at a low level between rewrite graphs is typically performed by checking for equal contributions of the two graphs for a given pregnance (same coordinates on the dimension associated to the pregnance) and matching the objects in their respective graphs against pre-defined patterns.

4 Executive

The Ikon Flux executive is similar in concept to a distributed virtual machine: it interprets objects and performs the rewritings (also called *reductions*) regardless of their location in a cluster. This section describes the sub-systems that constitute the Ikon Flux executive.

4.1 Architecture

Performing in real-time deep pattern matching over a massive[11] amount of programs is computationally intensive but not intractable. Ikon Flux has been designed to take advantage of distributed computing resources and version 1.6 has been implemented on a cluster of standard PCs[12] running RTAI[13] and interconnected through RTnet[14].

For many reasons – essentially, performance concerns and intractability of axiomatizing industrial-grade components - the executive has not been written in the Ikon Flux language and no model of its operational semantics has been made available at the code level. In other words, the executive is not meant to evolve.

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

[12] Pentium IV 3.2GHz, 4GB RAM.

[13] A real time kernel extension for Linux - see www.rtai.org

[14] A real time network protocol stack – see www.rts.uni-hannover.de/rtnet

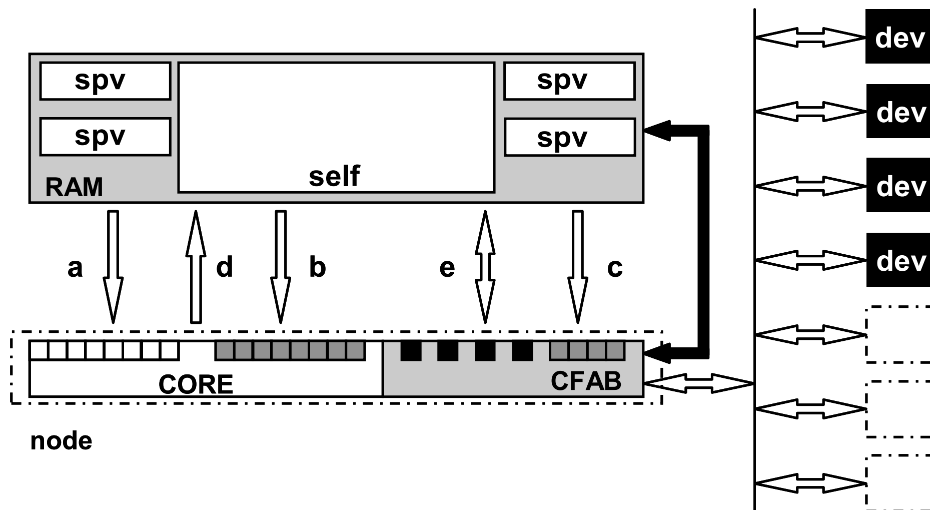


Figure 9 – Architecture of a computing node. *Objects are constructed from operators (a), CORE functions (b) and CFAB functions (c). They also interact with external sub-systems (e). The executive automatically injects objects encoding reduction events (d). Objects live in RAM, kept consistent across the nodes by the CFAB (black arrows). The CFAB also updates the I/O stacks (black squares) of the external sub-systems (dev) that usually run on separate machines.*

Core sub-system. This is the reduction engine. It performs reductions in cycles of five main steps:

1. the core sub-system deletes sys-objects whose resilience dropped down to zero; it notifies the outcome of every reduction operation performed during the previous cycle, including the execution of core functions.
2. external sub-systems are polled for inputs, i.e. inputs are popped out of their output stacks.
3. sys-objects are fed as input to programs performing modifications on control values. Control values are used to select both the inputs and the programs. Modifications are averaged.
4. sys-objects are fed as inputs to programs performing productions, accordingly to control values updated in step 3.
5. sys-objects produced during the cycle and representing function calls on external sub-systems are pushed in the sub-systems input stacks. Other products are injected in the sys-realm for the next iteration. These include runtime reflective data (application and process objects).

The core sub-system offers three main categories of functions: (1) synthesis (injection, modification and assignment of control values, respectively **inj**, **mod** and **set**), (2) contextualization (**inv** to compute inverse models, **ctx** to project sys-objects according to their contribution to the achievement of processes) and, (3) monitoring functions (**min** and **max** monitoring the maximal, respectively minimal values of a function applied to objects matching a specific pattern).

System clock. The system clock provides the time and reduction count (functions **now** and **rdc**) at any point in a reduction cycle from the booting time of the system (function **btt**). In addition, the system clock injects a tick object (**tck**) in each reduction cycle. The time resolution is 1 ms.

Computing fabric. This sub-system represents the component that distributes computation across the cluster. It implements a MIMD scheme where memory is replicated in every cluster node (*n x n* connectivity, where *n* is the node count). Each node is dedicated to the execution of selected subsets (work list) of the programs in the system. The programs resulting from these reductions are appended to the node's work list.

The computing fabric determines the depth of reduction cycles as the number of cycles before updating, cluster-wide, the reduction results obtained locally by core sub-systems (this reduction depth is a parameter that can be modified with the function **rdp**). Accordingly, external sub-systems I/O are only performed at memory update times.

Each node maintains a representation of each other work list (as a part of the replicated memory) and load balancing is achieved by making the content of the working lists even, under the assumption that every node is identical – the norm in high performance computing clusters. Each node can define a reduction budget (expressed in time quanta via the function **rdq**). Load balancing is performed by evaluating the actual time consumption against the available budget, and triggering if needed content migration between work lists.

The computing fabric also ensures that the nodes are synchronized, i.e. that the time stamps are consistent across the network and that reduction cycles are performed synchronously at the granularity of the reduction depth.

The computing fabric handles the connection between the core sub-systems and the external sub-systems – typically running on separate machines.

The computing fabric notifies to core sub-systems the events related to distributed resource management like (un)loading sub-systems, nodes joining (or leaving) the cluster.

Supervision sub-systems. Supervision sub-systems belong to a generic class of sub-systems, representing reduction sub-systems as sets of objects that can read and write in the global memory but cannot be written by general sys-objects. Supervision sub-systems are used to hold probes for debugging / profiling purposes, but can also – in a general way – hold any non evolvable code needed by the developer to perform direct control over the activity of the system.

As for reduction systems, supervision sub-systems are organized in realms, and this is in fact the case for the self and all other entity in the system.

The supervision sub-system class provides code-handling functions: a code loading function (**ldr**), a swapping function (**swp**), and a function for controlling reduction (**rct**), as usually exhibited by debuggers (start, stop, halt, resume, etc).

4.2 Extensibility

As introduced earlier, Ikon Flux can be extended by developers intending to define additional axioms or initial ontologies for particular systems in various ways. The following list summarizes them:

- operators and data types: operators must come with rewrite rules defining their related equational logic for inverse modeling purposes (**erw**, **inv**);
- markers: to define new ontologies, or, more generally, new categories (**markers**). Operators can be defined to interpret these new markers.
- external sub-systems: to implement the connection with heterogeneous componentry. Defining new external sub-systems leads to defining new operators to identify their functions and also possibly, new object types and constants. Developers shall implement their sub-systems so that they (the sub-systems) provide runtime reflectivity in the same fashion the core sub-system does.
- core functions: future versions will propose a programming model for extending the set of core functions by user-defined code (plug-ins).

An API has been defined to frame the development of extensions (this API is not described in this report).

4.3 Performance

Loki was built on Ikon Flux 1.6 and was running on a cluster of 16 machines (not counting external sub-systems). It was constituted by nearly 300 000 sys-objects among which 20% where intense/active enough to trigger reductions. With a reduction depth of 4, the memory update frequency was 4Hz for an average of 800 reductions per second and per node. Our measurements of Loki's performance show that under constant load, rewriting speed scales fairly well with the number of processors: for a load of N sys-objects distributed over n nodes, adding one node yields a new load per node of roughly $N/(n+0.8)$, instead of the ideal value $N/(n+1)$. However, the measurements also show that the admissible load is limited by inadequate networks and protocols (low bandwidth, high latency, very poor scalability): from 8 to 16 nodes under a constant load, the memory update frequency did not catch up with the

improvement of reduction speed. As a result, a new implementation (for version 2.0) is planned, targeting modern hardware (multi-core processors, RDMA[15] over Infiniband).

5 Related Work

Ikon Flux has been built to enable the construction of structurally autonomous systems: systems that can improve their utility function in real-time and in open-ended environments. The architecture described here is a platform for experimenting new designs and new engineering methods to build truly autonomous systems, the theory of which still remains to be conceived. The emphasis is put on the parallel execution of a multitude of programs in real-time and on the construction of reflective knowledge to control what is the essence of the targeted systems: targeted code evolution. Very little work has been done so far to support the engineering of systems of this class. However a few architectures stand up at the upper boundary of behavioral autonomy, having the potential to be brought to increased levels of adaptivity. Here is a succinct review of three salient examples; I also indicate the common grounds that these approaches share with Ikon Flux.

AKIRA (Pezzulo and Calvi 2005, 2007) is a schema–based computational substrate that controls the execution of individual schemas with a connectionist activation network, borrowed from DUAL (Kokinov 1997). Essentially, Ikon Flux shares with AKIRA (1) the homogenous approach (in AKIRA everything is a schema instead of a program), (2) the control on a global scale, (3) support for the cooperation of sets of programs (called *bands* and *hordes* in AKIRA) and, (4) support for the modeling of program interaction (AKIRA also uses forward and inverse models). The three main differences between thi sworka nd Ikon Flux are that (1) AKIRA does not focus on generativity, (2) it does not support runtime reflectivity, i.e. the execution of schemas is not monitorable at the substrate level and, (3) it does not support the modeling of its own components - schema code is opaque and no specification for it is made available. In a general way, the granularity of a schema is too coarse to allow the dynamic modeling of their operation. Their non-explicit structure and the unrestricted presence of internal states and side effects limit in practice the scope of online reasoning to *surface* operations, in line with the tenets of the behavioral approaches to modeling. On the technical side, AKIRA has not been engineered to scale up with the computation load. It features client/server capabilities but genuine distribution of the executive and of the load for real-time performance was not part of its requirements. From a technical perspective, doubtlessly, AKIRA is well-born and its shortcomings can be addressed without changing dramatically its nature. This means reducing the granularity of schemas, supporting schema generativity, improving the reflectivity and transparency of both schema structure and processes and finally, developing schemas to measure and control the formation of bands and hordes.

NARS (Wang 2004, 2007) results from an approach that focuses on reasoning in real-time with insufficient knowledge and resources in changing environments. NARS – Non-Axiomatic Reasoning System - has been designed form the beginning to support the production of experience-grounded semantics and in that respect, Ikon Flux adopts similar views. On the technical side, NARS puts primarily its emphasis on logic to represent and infer knowledge and encode the control of real-time tasks at the substrate level - structures are made explicit and reflective: NARS has been designed to express the tasks' operational semantics as executable models and to reason about them. This makes NARS eminently suitable for developing models of internal processes experimentally. On the theoretical side, NARS is among the very few systems today that emphases a generative approach towards structure building and grounding system operation in light of goal achievement. Wang adopts a theoretical stance of envisioning systems as holistic organizations - Ikon Flux follows a similar perspective although for entirely different reasons, briefly, for architectural and engineering reasons rather than in light of anthropo-centric considerations. In any case, the *practical* question of how to observe and control a large and complex system *as a whole* has by and large not received much attention. To achieve such a global control, one needs to evaluate the resources involved between the occurrence of salient inputs and the generation of salient effects. In NARS, the execution of tasks is essentially reflected by means of implicit and indirect effects on other tasks (selection, available CPU, response times, etc.). In order for a system to reason about the development of one of its processes – i.e. its incremental construction as task graphs and its execution – one would need a *trace* of its causes and effects in the system over time. In that respect, NARS would most likely benefit from the addition of means to express the global and explicit operational semantics of dynamic organizations of tasks.

---

[15] Remote Direct Memory Access. Switched interconnects like Infiniband or Rapid-IO enable the implementation on commodity hardware of mainframe-style designs for real-time systems, in the fashion of crossbar organizations: streams of computation flowing within a unified memory from/to a multitude of processors.

The architecture developed by Cardon et al. (Cardon 2003, Cardon et al. 2005, Campagne 2005) is one of the most advanced architecture implemented today for building actual systems endowed with a fair degree of behavioral autonomy. It follows the (massively) multi-agent paradigm, as does AKIRA. Cardon's architecture is a massive organization of agents organized to form a single "mind". This organization is controlled in real-time and at a global scale using morpho-analysis – also based on Thom's general theory of models (Thom 1972 and 1999). Agents in Cardon's architecture do not come with a fine-grained model of their function and execution. Only the general aspects of the agents and their population is captured and used for control: these are the contributions of teams of agents to the achievement of another team's work. But modeling the functioning of the agents themselves does not seem to have received any explicit support. Compared to Ikon Flux, this architecture offers superior handling of the global trajectory of the system in the state space, but unfortunately does not go deep enough to enable fine and accurate control at arbitrary scales of implementation, from the global morphogenesis process down to the code level, i.e. the level where are defined the agent's individual behaviors and internal observation/control schemes. Lastly, the agents responsible for the morpho-analysis can be generated dynamically, but this is not the case for the rest of the system: the agents that actually perform the tasks in the environment are still individually hard-coded. The latter two points – limited scope for modeling and limited generativity - indicate the current limitations on adaptivity in this approach. These limitations, however, pertain only to the *scope* of implementation of the key concepts, nothing that cannot be solved by subsequent refinements of the present design.

The works discussed in this section represent fairly well the state-of-the-art of architectures for behaviorally autonomous systems at the point of this writing. It is no doubt possible to introduce additional features – where relevant - to these existing systems to endow them with increased levels of autonomy without breaking down their core concepts. These improvements would include essentially:

- *down-sizing the grain of the components* that constitute the computational substrate,
- enabling the *dynamic construction of the operational semantics* of these components in the same substrate,
- allowing the *global observation, modeling and control at arbitrary scales* of implementation and,
- provisioning, at the executive level, for the *reflective and real-time execution* of component organizations of *very large sizes*. On the latter point – large scale organizations: only a significant amount of complexity is adequate to manifest and therefore to experiment meaningful responses to the problem of global control with acceptable response times in *real-world systems*.

Reciprocally, Ikon Flux would certainly benefit from implementing a morphogenetic perspective on the dynamics of global observation and control along the lines described by Cardon et al. Ikon Flux is naturally very well suited for this approach as processes are already represented in topological spaces in a way that accounts for the collaboration/competition of their respective constituents. Morphogenetic observation/control methods will be considered for integration in future versions of the executive.

6 Conclusion

In the domain of computational systems, autonomy can be operationalized by the concept of self-programming, for which Ikon Flux is an abstract type: a dynamic and self-referential architecture generating its own structures and processes in real-time, both grounded mutually in high-dimensional topological spaces.

An actual system has been built from this proto-architecture, using experimental engineering methods. Building the bootstrap segment of such a system is the real difficult part: to my knowledge, there exists today no actual methodology for the *principled engineering* of evolutionary systems, and this is an open issue that calls for investigation along two main lines. First, it is leveraging prior knowledge and meta-knowledge by coding analogy making programs and abstraction building programs - essentially along lines like the ones described by Dormoy and Kornman (1992). These programs are used to infer goals and plans - by generating new heuristics from the ones provided in the bootstrap segment – and also to unify local models into more general ones. The second and more decisive research avenue is identifying and fostering the formation and the transformation of high-level structures, functions and processes. 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 keep the system complexity at a reasonable level as it expands by producing new knowledge, and (4) to identify and optimize existing rewrite graphs. In other words, observing

and controlling the formation of high-order structures is a necessary step towards endowing a system with self-organization properties.

Pursuing the preliminary goal of designing a vocabulary and a grammar to identify, express and design the dynamics of structural transformations, I resorted so far to injecting some “architectural” code in the bootstrap segment: these are experimental programs and models able to identify and construct instances of classes of high-level organizations. These classes are defined as patterns of process interaction, technically, the mutual projection of realms on some others:

- *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 (set of) pregnance(s).

The identification of high-order structures such as these has been encoded in the Ikon Flux language relatively easily: this is always the case when one knows beforehand what to look for. In that respect, this method of identification can be viewed as an *allonomic* measurement procedure. In the long term, the method in question will most likely restrict the scope and depth of system self-improvement. At this stage of development, most of the bootstrap segment is to be kept away from evolution, notably architectural code and internal drives. Loki‘s bootstrap code represented roughly 30% of the whole system, which translated into a significant amount of tedious manual labor. Instead of relying on a static typology, this calls for anchoring the development of bootstrap code in a model of the *dynamics* of the formation of structures. Even in the - relatively - simplified case of Loki, to control the formation in both general and predictable ways remains an open issue.

Harnessing the emergence of high-order organizations in a general scalable way calls for new engineering methodologies and as I have argued here, these must be based on a *process-oriented* and *generative* approach. To this end I am currently investigating the possibility of modeling self-programming processes using Goldfarb’s (2005) Evolving Transformation System.

References

Ancel L.W., Fontana W. (2000). *Plasticity, Evolvability and Modularity in RNA*, Journal of Experimental Zoology. 288:242–283.

Campagne J.C. (2005). *Morphologie et systèmes multi-agents*. Ph.D. thesis, Université Pierre et Marie Curie, Paris.

Cardon A. (2003) *Control and Behavior of a Massive Multi-agent System.* In Truszkowski W., Rouff C., Hinchey M. eds. *Workshop on Radical Agent Concept* 2002, LNAI 2564, pp. 46-60. Springer-Verlag Berlin Heidelberg.

Cardon A., Campagne J.C., Camus M. (2005) *A self-adapting system generating intentional behavior and emotions*. In *Workshop on Radical Agent Concept* 2005, NASA Goddard Space Flight Center.

Dormoy J.L., Kornman S. (1992). *Meta-knowledge, autonomy, and (artificial) evolution: Some lessons learnt so far*. In Varela F.J. and Bourgine P. eds. *Toward a Practice of Autonomous Systems* Proceedings of the 1st European Conference on Artificial Life. MIT Press (A Bradford Book), 1992: 392-398.

Fontana W., Buss L.W. (1996). *The Barrier of Objects: From Dynamical Systems to Bounded Organizations*. In Casti J., Karlqvist A. eds. *Boundaries and Barriers*. pp.56–116, Addison-Wesley.

Froese T., Virgo N., Izquierdo E. (2007). *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, in press.

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

Hoover *M.L. (1974) Meyerhold: The art of conscious theater* University of Massachusetts Press, 1974

Kokinov, B. (1997). *Micro-level hybridization in the cognitive architecture DUAL*. In R. Sun & F. Alexander eds., *Connectionist-symbolic integration: From unified to hybrid architectures*. Hilsdale, NJ: Lawrence Erlbaum Associates.

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

Meyerhold V. (1969). *Meyerhold on Theatre*, trans. and ed. by Braun E. Methuen, London.

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

Paun G. (2002). *Membrane Computing. An Introduction*, Springer-Verlag, Berlin.

Pezzulo G., Calvi G. (2005). *Dynamic Computation and Context Effects in the Hybrid Architecture AKIRA*. In Dey A., Kokinov B., Leake D., Turner R. eds. *Modeling and Using Context* 5th International and Interdisciplinary Conference CONTEXT 2005*.* Springer LNAI 3554.

Pezzulo G. Calvi, G. (2007). *Designing Modular Architectures in the Framework AKIRA*. In *Multi-agent and Grid Systems*, 3:65-86.

Pollock J. (2001). *Defeasible reasoning with variable degrees of justification*. In Artificial Intelligence, Vol. 133, Nos. 1-2, p. 233-282.

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

Rocha L.M. (2000). *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.

Sanz R., López I., Hernández C. (2007). *Self-awareness in Real-time Cognitive Control Architectures*. In *AI and Consciousness: Theoretical Foundations and Current Approaches*. AAAI Fall Symposium 2007. Washington, DC.

Schmidhuber J. (2004). *Optimal Ordered Problem Solver.* Machine Learning, 54, p. 211-254.

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

Spector L., Klein J., Keijzer M. (2005). *The Push3 execution stack and the evolution of control*. In *Proc. Genetic and Evolutionary Computation Conference (GECCO 2005)*, p. 1689–1696.

Thom R. (1972). *Structural Stability and Morphogenesis*. Reading, MA: W. A. Benjamin.

Thom R. (1990). *Semiophysics: A Sketch*. Redwood City: Addison-Wesley.

Thom R. (1999). *Paraboles et catastrophes*. Flammarion.

Varela F.J., Maturana H.R., Uribe R. (1974). *Autopoiesis: the organization of living systems, its characterization and a model*. In: *BioSystems*, 5:187–196.

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

Varela F.J. (1992). *Autopoiesis and a Biology of Intentionality*. In McMullin B., Murphy N. eds. *Autopoiesis and Perception* A Workshop with ESPRIT BRA 3352 (pp.4-14). Dublin.

Wang P. (2004). *Toward a Unified Artificial Intelligence.* AAAI Fall Symposium on *Achieving Human-Level Intelligence through Integrated Research and Systems.* p. 83-90, Washington DC.

Wang P. (2007). *The Logic of Intelligence*. In *Artificial General Intelligence*. Springer Verlag.

Wolpert D.M., Kawato M. (1998) *Multiple paired forward and inverse models for motor control*, Neural Networks 11, pp. 1317–1329.

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

Annex – Language Definition

A1 Overview

The Ikon Flux language defines seven categories of constructs:

- *Atoms*: constants, blocks (lists of atoms), identifiers, variables and code fragments. The latter are the basic sub-structures that are composed together to form system objects – sys-objects (`sys`). They are neither the input nor the product of rewriting – only sys-objects are.
- *Directives*: instructions to specify how the interpreter shall handle code. Directives are not subject to rewriting.
- *Operators*: symbols defining operations to be performed on a list symbols; an operator followed by such a list constitutes a fragment. Operators are implemented in the executive and their semantics cannot be modified at runtime. Operators can be user-defined. Operators fail gracefully when applied to ill-formed fragments.
- *Object types*: symbols defining *static* types. A type defines a specific fragment structure and its (axiomatic) semantics – exception: sub-systems are axiomatic objects and have no structure. Objects can be user-defined.
- *Sub-systems*: symbols representing either axiomatic systems (external, i.e. application dependent or internal, i.e. the components of the executive) or explicit systems, i.e. abstractions denoting phenomena (external, i.e. phenomena observed in the world, or internal, i.e. phenomena observed in the state space). External axiomatic sub-systems can be user-defined. Explicit sub-systems (called *entities*) can be dynamically instantiated.
- *Sub-system functions*: symbols representing function entry points in a given sub-system. Sub-system functions cannot be rewritten as they are implemented in an non-explicit form. A function symbol followed by actual arguments is yet another example of fragment. Sub-system functions can be user-defined. Sub-system functions fail gracefully when applied to ill-formed fragments.
- *Marker classes*: symbols representing ontologies, i.e. *dynamic* object types. They define fragment structures but not their semantics – their rewriting and use by programs and models does. Marker classes can be user-defined. Marker classes can be dynamically instantiated.

Operators, objects types, sub-systems, sub-system functions and marker classes are represented by numerical *opcodes*; However, for the sake of readability, when these elements are built-in the executive, they are identified by trigrams and/or by non numerical symbols (aliases).

A1.1 General syntax

From this point onwards, the following syntactic elements are used to define the language constructs:

key	*description*
`in italic`	optional
`...`	list of atoms / fragments, etc, whatever
`<expression>`	syntax of `expression` defined separately
`<arg=value>`	optional argument with (optional) default value

Atoms

symbol	*description*
`nil`	“nothing”, polysemic
`()`	block, generic fragment
`( ... )`	nested definition of a fragment, block or generic fragment
`a b c`	code fragment composed of atoms `a`, `b` and `c`
`o: f`	definition of an object / fragment name mapping a fragment `f`
`:v`	definition of a variable `v` (in a pattern) mapping any fragment
`::v`	definition of a variable `v` (in a pattern) mapping any rest of fragment
`:`	anything (wildcard, i.e. unreferenced variable in a pattern)
`::`	the rest of the fragment can be anything (in a pattern)

<table>
<tr><td>name:</td><td>fragment name</td></tr>
<tr><td>v:</td><td>right-hand reference to a variable (equational rewrite rules only)</td></tr>
<tr><td>v, o</td><td>reference to a variable v, to an object o</td></tr>
<tr><td>this</td><td>reference to the address of an object from inside that object (C++ fashion)</td></tr>
<tr><td>-10, 1.234+56</td><td>numbers (IEEE754, 64 bits). 1.234+56 means 1.234 10+56. Infinite numbers are supported. NaN is coded nil</td></tr>
<tr><td>+inf, –inf</td><td>positive infinity, negative infinity</td></tr>
<tr><td>//, ///</td><td>comment, comment block opening and ending</td></tr>
<tr><td>mk.id</td><td>reference to a marker of class id</td></tr>
<tr><td>ent.id</td><td>reference to an entity named id</td></tr>
<tr><td>spv.id</td><td>reference to a supervision sub-system named id</td></tr>
<tr><td>dev.id</td><td>reference to an external sub-system named id</td></tr>
<tr><td>sys.id</td><td>reference to an internal sub-system named id</td></tr>
<tr><td>mk.:</td><td>reference to any marker (also holds for ent, spv, sys and dev)</td></tr>
<tr><td>mk.:x</td><td>definition of a variable x mapping any marker (also holds for ent, spv, sys and dev)</td></tr>
<tr><td>mk.x</td><td>reference to a variable mapping a marker (also holds for ent, spv, sys and dev)</td></tr>
<tr><td>|</td><td>“contrary” meta-operator, polysemic. Applies to some operators and objects, but neither to markers nor sub-system functions.</td></tr>
</table>

Directives

directive	*description*
if, then, else, endif	control the code definitions
macro	defines a syntactic substitution
load	loads source code files
region	defines a group of code
in	defines membership of code relatively to regions

Built-in operators

<table>
<tr><th>operator</th><th>alias</th><th>contrary</th><th>description</th></tr>
<tr><td>equ</td><td>=</td><td>|=</td><td>tests for equality</td></tr>
<tr><td>sor</td><td>></td><td><=</td><td>tests for strong order relationship</td></tr>
<tr><td>wor</td><td>>=</td><td><</td><td>tests for weak order relationship</td></tr>
<tr><td>add</td><td>+</td><td>-</td><td>addition</td></tr>
<tr><td>mul</td><td>*</td><td>/</td><td>multiplication</td></tr>
<tr><td>new</td><td></td><td>del</td><td>instantiation</td></tr>
<tr><td>eva</td><td></td><td>|eva</td><td>evaluation</td></tr>
<tr><td>red</td><td></td><td>|red</td><td>reduction</td></tr>
<tr><td>scn</td><td></td><td>|scn</td><td>scan</td></tr>
<tr><td>mrg</td><td></td><td>N/A</td><td>merge blocks</td></tr>
<tr><td>spl</td><td></td><td>N/A</td><td>split blocks</td></tr>
<tr><td>pow</td><td></td><td>log</td><td>power</td></tr>
<tr><td>exp</td><td></td><td>ln</td><td>exponent</td></tr>
<tr><td>dis</td><td></td><td>N/A</td><td>distance</td></tr>
</table>

Built-in object types

object type	*description*
pgm	program
spc	pattern specification
pre	pregnance
app	application (of a function, a model, a pregnance)
sln	saliency
ifn	implicit function

efn	explicit function
pro	process
fmd	forward model
imd	inverse model
rlm	realm
sys	sys-object
dev	external sub-system
spv	supervision sub-system
ent	entity
tck	clock tick
erw	equational rewrite rule

Built-in sub-systems and functions

sub-system	*alias*	*function*	*description*
sys.0	core		core reduction engine
		inj	injection of an object
		mod	modification of control values
		set	assignment of control values
		get	read control values
		inv	computation of an inverse model
		ctx	projection of sys-objects on a realm
		min	monitoring of the maximum of a target application
		max	monitoring of the minimum of a target application
		avg	monitoring of the average of a target application
		rnd	random number generator
sys.1	clk		system clock
		btt	gets the boot time
		now	gets the current time
		rdc	gets the current reduction count
sys.2	cfab		distributed computation fabric
		rdp	sets the reduction depth
		rdq	sets the reduction quanta
spv.x			supervision sub-systems
		ldr	loads code
		rct	controls the runtime of the supervision sub-system
		swp	swaps code from/to disk
ent.0	self		axiomatic reference to the system. No functions are pre-defined; they have to be constructed dynamically or manually (bootstrap code).

Built-in marker classes

marker class	*alias*	*description*
mk.0	mk.pja	projection, activation semantics
mk.1	mk.pji	projection, intensity semantics
mk.2	mk.ins	instance
mk.3	mk.hyp	hypothesis
mk.4	mk.sim	simulation
mk.5	mk.prd	prediction
mk.6	mk.xet	process execution time
mk.7	mk.min	minimal application value
mk.8	mk.max	maximal application value
mk.9	mk.avg	average application value

Code fragment

fragment_name: *opcode <data>*

`opcode`: identifies an operator, an object, a function, a sub-system or a marker class. If omitted in a nested definition, the fragment is:

- a block or,
- a general fragment with no built-in semantics.

`data`: list of atoms.

Notes

The implicit conversion of a block in its first element will be performed if needed; the implicit conversion of a fragment in a block containing it will also be performed if needed. This need can arise during the evaluation of fragments lead by the `red`, `mrg`, or `spl` operators.

Examples

```
<= :x (/ y ( + :z w)) // :x <= y/(:z+w)

|= a frag1: (red ( ... ) (pgm (spc in:(sys self ::) nil) ((nil in)))) // when successfully
evaluated, frag1 points to the result - here (sys self ::), to nil otherwise
```

Notes

Variables are pointers to fragments formally defined within pattern specifications. Variables defined inside a sys-object are not visible outside, but are everywhere inside.

Separators: in { ' ', '\n', '\tab', '\backtab', '\'}. They are not required after nor before (or), neither after object/fragment definitions. NB: some of the separators have side effects, see 0 below.

`app` is implicit when the opcode leading a fragment maps to a function, a pregnance, a (forward/inverse) model.

`red` is implicit when the opcode leading a fragment maps to a pattern specification.

Identifiers (fragment names and variables) are case sensitive and can contain any character / strings but those in:

{ separators, '(', ')', ':', "::", '|', '/' }

As a general rule, identifiers cannot be written as any operator / object / function / marker / fragment could be. Identifiers are independent from directives.

Literal symbols (alphanumerical character strings) are hosted in external sub-systems, referenced to by numerical identifiers. In the source code, literal symbols are mere numbers.

A1.2 Code syntactic form

Code is defined using indents to limit parenthesis explosion.

Examples

The following two examples are legal and describe the same code, i.e. the code introduced in section 3.1.

```
pgm1: pgm
          spc (sys :x :mk1 ::)
              (red mk1
                   pgm
                       spc in1: (sys (mk.ins x :a) :mk2 ::)
                           (red mk2
                                pgm
                                    spc in2: (sys (mk.ins a :b) :int :act :res :) nil
                                    ((nil in2))
                           )
                       ((nil in1))
              )
          ((nil
                inj (core (sys (mk.ins x b) act int res)))
          )
// this empty line is required

pgm1: pgm
           spc (sys :x :mk1 ::)
               () // means generic fragment, or block
                   red mk1
                       pgm
```

```
                          spc in1:(sys (mk.ins x :a) :mk2 ::)
                              ()
                                 red mk2
                                     pgm
                                         spc in2:(sys (mk.ins a :b) : :int :act :res :) nil
                                          () (nil in2)
                          ()
                              nil in1
          ()
             () nil
                inj (core (sys (mk.ins x b) act int res))
// this empty line is required
```

A1.3 Source code organization

General structure for application code:

List of directives.

List of object definitions.

List of applications (ex: object injections).

Setting of the reduction depth and quanta (applications of `rdp` and `rdq`).

Structure for equational rewrite rules:

List of directives.

List of object definitions.

List of rewrite rules definitions.

Notes

The following are predefined and pre-loaded:

sys-realm: `srm`

sub-systems: `core`, `clk`, `cfab`, `self`

The equational rewrite rules defined for the built-in operators are also pre-loaded.

Object definitions and code management directives shall not be mixed, i.e. directives shall be invoked in their own lines.

Directives can be found anywhere; they are optional.

Applications (`app`) can also be found anywhere provided the fragments they refer to are defined.

Redefining different code with the same identifier overwrites the previous definition and remaps existing references to the latest definition: allows forward incomplete definitions (C++ style).

A2 Language constructs

A2.1 Directives

LOAD - Code loading directive.

`load <file id>`

`file id`: integer.

Loads a source file and translates it into a binary form, or loads binary code.

Notes

Files are identified by integer values (e.g. symbolic links to actual files).

Binary code is produced by `swp`.

REGION - Region definition directive.

`region <id>`

`id`: an integer identifying the region.

Regions define logical groups of hand-crafted code.

Notes

Region are provided for code management purposes. They are not objects and are not visible in the sys-realm nor can they be generated dynamically. However, fragments can control the (un)loading of regions.

IN - Region membership directive.

`in <id list>`

`id list`: integers identifying regions.

Objects defined after a membership directive are members of the specified regions, until a new membership directive is found.

Notes

Objects can belong to several regions. In that case they are load when the first region is loaded and unloaded when the last region is unloaded.

MACRO - Syntactic development directive.

`macro <expression>` *`<fragment>`*

`expression`: `name` or `(name` *`<arglist>`*`)`.

`arg`: any fragment.

Specification of syntactic developments.

`|macro` undefines the syntactic development.

Examples

Predefined:

```
macro self ent.0
macro core sys.0
macro clk  sys.1
macro cfab sys.2
macro _now (now (clk))
macro _rdc (rdc (clk))
macro _btt (btt (clk)),
macro _rdp (rdp (cfab))
```

User-defined:

```
macro (find mk.class a b block1 guards block2 act int res ijt)
            red block1 (pgm (spc (in: sys (mk.class a b) :block2 act int res ijt) guards) ((nil
in)))
macro run_once   -1
macro match_once -2
macro run_match_once -3
macro _int 0
macro _act 1
macro _res 2
```

Usage:

```
pgm1: pgm
          spc (sys :x :mk1 ::)
              ()
                  (find mk.ins :x :a mk1 ((find mk.ins mk2 a :b :act :int :res :)) mk2 ::)

          ()
              nil /// empty guard block /// (inj core (sys (mk.ins x b) act int res))
```

IF THEN ELSE ENDIF - Source control directive.

`if <macro> then <def1>` *`else <def2>`* `endif`

`macro`: definition of a syntactic development.

`def1`, `def2`: lists of definitions / applications / directives.

Controls the definitions / directives in a source file. If there exists a definition for `<macro>` then `<def1>` is processed, else `<def2>`.

Notes

Similar to the C++ pre-processor directives #ifdef ... #else ... #endif.

A2.2 Operators

= Equivalence operator.

```
= <t1> <t2>
```

t1: any fragment.

t2: any fragment.

Checks for equivalence of t1 and t2. Returns |nil if successful, nil otherwise.

|= checks for the non equivalence of t1 and t2. Returns |nil if successful, nil otherwise.

> Strong order relationship operator.

```
> <t1> <t2>
```

t1: any fragment.

t2: any fragment.

Checks for the strong order relationship between t1 and t2. Returns |nil if successful, nil otherwise.

|> is noted <=; checks for the weak order relationship between t2 and t1. Returns |nil if successful, nil otherwise.

>= Weak order relationship operator.

```
>= <t1> <t2>
```

t1: any fragment.

t2: any fragment.

Checks for the weak order relationship between t1 and t2. Returns |nil if successful, nil otherwise.

|>= is noted <; checks for the strong order relationship between t2 and t1. Returns |nil if successful, nil otherwise.

+ Addition operator.

```
+ <t1> <t2>
```

t1: any fragment.

t2: any fragment.

Performs the addition of t1 and t2, returns the result.

|+ is noted -; performs the substraction of t2 from t1, returns the result.

***** Multiplication operator.

```
* <t1> <t2>
```

t1: any fragment.

t2: any fragment.

Performs the multiplication of t1 and t2, returns the result.

|* is noted /; performs the division of t1 by t2, returns the result, nil if t2=0.

NEW Instantiation operator.

```
new <t>
```

t: mk, ent.

Depending on the argument, creates a new marker class, or a new entity, returns the new object.

|new is noted del; deletes its argument and its related fragments: when a marker class is deleted, the sys-objects instances of this class are deleted too. del works on any object, not only the types specified for new.

Notes

Deleting an object triggers a garbage collection notification.

EVA Evaluation operator.

`eva <t>`

`t`: any fragment.

Performs (forces) the evaluation of its argument.

`|eva` prevents the evaluation.

Notes

Evaluation is always implicit in the generation of productions (programs, functions, models). It is also implicit when a pattern specification is tested against an object: the object is evaluated step by step following the structure of the pattern.

The attempt to evaluate ill-formed fragments fails gracefully, returning `nil`.

RED Reduction operator.

`red <b> <t>`

`b`: a block or a single fragment.

`t`: a fragment, pattern specification, program or block of programs / pattern specifications.

Performs the reduction of `b` by `t`, returns a block containing the results. Reduction is performed as follows:

- if `t` is a program, then all the fragments in `b` are evaluated against the program pattern specification, the productions, if any, are appended to the result block.
- if `t` is a pattern specification, then all the fragments in `b` are evaluated against it and they (the fragments in `b` that match) are appended to the result block.
- if `t` is a fragment (ex: `(= x nil)` where `x` is a reference to a variable), then all the fragments in `b` are evaluated against it and the result is appended in the result block.
- if `t` is a block, then all the fragments in `b` are evaluated against each fragment in `t`, as described above.
- if `b` is a single fragment, then reduction is performed as described on b instead of the content of `b`.

`|red` performs the anti-reduction (see `|pgm`, `|pre`). `t` - or fragments in `t` if it is a block - are appended to the result block if no fragments in `b` - or `b` if it is a single fragment - evaluates successfully.

SCN Scan operator.

`scn <pattern> <production>`

`pattern: spc <fragment> <guards>`

`production`: any fragment.

Evaluates every sys-object against the pattern, returns a block containing the productions or `nil` if no match has been detected. This is the standard reduction, i.e. controlled by the intensity values of input objects, and the activation value of the program executing `scn`.

`|scn` checks if no sys-object matches the pattern, returns the production if successful, `nil` otherwise.

Notes

The scan operator allows computing new patterns during reduction and checking the sys-objects against these patterns during the *same* reduction cycle. Without `scn`, one would have to generate a program holding a new pattern, and check the sys-objects against this new pattern during the *next* reduction cycle.

MRG Merge operator.

`mrg <b1> <b2>`

`b1`: a block or a fragment.

`b2`: a block or a fragment.

Returns one block, having the content of `b1` and of `b2`. If `b1` is a single fragment, `b1` is appended to `b2` in the result, and reciprocally. If `b1` and `b2` are single fragments, returns a block containing the two.

`|mrg` returns `nil`.

SPL Split operator.

```
spl <b> <pattern>
```

`b`: a block.

`pattern`: `spc <fragment> <guards>`

Returns a pair of two blocks: the first contains the fragments that match the pattern and the second, fragments that don't.

`|spl` returns `nil`.

POW Power operator.

```
pow <t1> <t2>
```

`t1`: any fragment.

`t2`: any fragment.

Performs the elevation of `t1` at the power of `t2`, returns the result.

`|pow` is noted `log`; computes the logarithm of `t1` in the base of `t2`, returns the result.

EXP Exponent operator.

```
exp <t>
```

`t`: any fragment.

Performs the elevation of *e* at the power of `t`, returns the result.

`|pow` is noted log; computes the Neperian logarithm of `t`, returns the result.

DIS Distance operator.

```
dis <t1> <t2>
```

`t1`: any fragment.

`t2`: any fragment.

Computes the distance between `t1` and `t2`, returns the result.

When `t1` and `t2` are numbers, returns `|t1-t2|`.

`|dis` returns `nil`.

A2.3 Objects

ERW Equational rewrite rule.

```
rule_name: erw <lht> <rht>
```

`lht`: left hand fragment, a specification (see `pgm`)

`rht`: right hand fragment, the production, i.e. a code fragment

Notes

Equational rewrite rules are not part of any sys-realm, i.e. they are not potential input objects, nor can they be generated dynamically (although nothing prevents that: this will be considered for future versions). They are used by the inference engine, which is not a sub-system but an internal part of the executive.

Examples

Key: a → b means "a is rewritten in b"; `y: <f>` is the name of the fragment `f` *and* a reference to the variable `:y`.

```
erw (spc z: (+ :x :y) nil) y: (- z x) // z=x+y → y=z-x

erw (spc (+ :x :y) nil) (+ y x) // x+y → y+x
```

```
erw (spc (+ (+ :x :y) :z) nil) (+ x (+ y z)) // (x+y)+z → x+(y+z)

erw (spc z: (* (:x :y) ((|= x 0))) y: (/ z x) // z=x*y → y=z/x when x ≠ 0
```

SYS Sys-object.

```
object_name: sys <t> <ref> <int> <act> <res> <ijt=_rdc+1>
```

t: explicit definition or variable or reference.

ref: reference to the reference block. This block is populated automatically with any sys-object pointing at object_name; in particular, ref blocks contain markers and projections.

int: intensity value.

act: activation value.

res: resilience value.

ijt: injection time, in reduction cycle count (homogenous to the return value of the function rdc).

Notes

When new sys-objects are specified for injection, the <ijt> is optional and by default, set to _rdc+1.

int and act are the result of a global computation (involving realms) at the time of the current reduction cycle. At injection time, projections on the sys-realm are created automatically: mk.pja <object> srm act and/or mk.pji <object> srm int.

For programs, res=-1 means kept alive until an input object triggers the program, kill at the end of the cycle (unless res has been increased in the meantime). For general input objects res=-2 means kept alive until it matches in a program (kill as for programs). In case of a program being an input, res=-3 will do as -1 and -2, and will switch to -2 upon triggering, or -1 upon matching.

Sys-objects can be injected in the future, i.e. _rdc+n, where n>1.

Duplicates of the same sys-object (same structure and same age) are eliminated automatically. The remaining original gets the maximum of the control values (minimum if negative) of the duplicates. Duplicates in blocks are also eliminated.

SPV Supervision sub-system.

```
spv_name: spv.id
```

id: integer identifying the sub-system.

Supervision sub-systems are instances of the reduction engine. They can read an write the memory, can be read from other sub-systems but cannot be written.

Notes

Supervision sub-systems are meant primarily for debugging/profiling purposes, but nothing prevents their usage as regular internal systems to build an entire system as an organization (e.g. recursive) of systems.

ENT Entity.

```
ent_name: ent.id
```

id: integer identifying the sub-system.

An entity is an axiomatic construct to identify phenomena, either internal (in the system) or external (in the world).

Notes

The self is defined as ent.0.

DEV External sub-system.

```
dev_name: dev.id
```

id: integer identifying the sub-system.

External sub-systems are aggregates of exogenous functions. Such a sub-systems provides an interface to communicate with the executive in the Ikon Flux language. An interface is an interpreter to/from binary invocation from/to application objects, plus user-defined object types and operators.

Notes

External sub-systems can be loaded / unloaded dynamically: this usually triggers notifications by the corresponding `dev` instance (developer's responsibility).

The core sub-system provides notifications of rewriting events by injecting new objects describing the reductions that have been performed: for example, "at time t, the occurrence of object T_0 triggered the production of object T_1 by object T_2". They constitute internal inputs, conveying immediate causality $((T_0,T_1) \rightarrow T_2)$. Developers can choose to implement causal loops in their external sub-systems, in the same fashion the executive does (runtime reflectivity), i.e. to have external sub-systems behave as if they were reduction engines. For example a locomotion device is expected to inject and relate the event "reached target" to the command "reach target" issued at t_0: "at time t_1, object "reach target" received at t_0 triggered the production of object "target reached" by device "locomotion"". Exceptions occurring during the execution of a command are also considered feedback information that are generally notified in the same way.

SPC Pattern specification.

```
spc_name: spc
    <pattern>
    ()
        <guard0>
        ...
        <guardn>
```

`pattern`: *pattern_name*: `<fragment>`

`fragment`: any fragment – generally containing variable definitions, but not necessarily.

`guard`: `<fragment>` or `<program>` or `<pattern specification>` or block of fragments, programs, pattern specifications.

A pattern is a partial description of the structure of any kind of object. It is also a procedural description (guards express conditions on the structure).

Pattern matching fails if at least one of the guards fails, i.e. returns `nil`.

Evaluation in pattern matching is lazy, i.e. performed from left to right and in case of failure, no further evaluation is attempted.

`|spc` is an anti-pattern. Matching against a fragment t will succeed if t does *not* match the pattern.

Notes

Applications of pattern specifications can be built in the same fashion as for functions: `(<pattern spec> <fragment>)`. This is equivalent to `(red <pattern spec> <fragment>)`.

Guards can specify interdependencies between variables in the pattern.

Examples

ex1

```
spc0: spc (sys (mk.ins :x :a) : : : :t) ((= t _rdc)) is equivalent to spc (mk.ins :x :a) : : : _rdc)
nil and to spc (:y :x :a) ((= x mk.ins))
```

`_rdc` will be called when `spc0` is evaluated, only if a successful match for `(sys (mk.ins :x :a) : : : :t)` is found (lazy evaluation).

```
pgm0:pgm <pattern spc>
        ()
            nil (inj (core (sys
                           pgm
                               spc (sys (mk.ins :x :a) : : : _rdc) nil // _rdc will be evaluated at
the time when pgm0 is triggered
                               ...
                           ...
            nil (inj (core (sys
                           pgm1: pgm
                               spc (sys (mk.ins :x :a) : : : (|eva _rdc)) nil // _rdc will be
evaluated at the time when the pattern specification of
                               ...                                         // pgm1 is evaluated
```

```
                            ...
```

ex2

```
pgm (spc (sys (imd : :target_state ::) ::) nil) // deep pattern: target_state is a pattern
specification in an inverse model held by a sys-object
    ()
        nil (inj (core (sys
                       pgm
                           spc (target_state some_args) nil // the specification will be some_args
if some_args matches target_state, nil otherwise
                           ...
                       ...
        nil (inj (core (sys
                       pgm
                           target_state // used as is
                           ...
```

PGM Program.

```
pgm_name: pgm
    spc <pgm-pattern>
        ()
            <guard0>
            ...
            <guardn>
    ()
        <production0>
        ...
        <productionm>
<sem count=-1>
```

```
pgm-pattern: input_object_variable_name: sys <t> <ref> <int> <act> <res> <ijt>
```

- `t`: explicit definition or variable (object) or reference or `:` or `::`
- `ref`: variable (ref block) or `:` or `::`
- `int`: intensity value or reference or `:` or `::`
- `act`: activation value or reference or `:` or `::`
- `ijt`: injection time value or reference or `:` or `::`

```
guard: <fragment>
```

```
production: ()
                <guard0>
                ...
                <guardn>
            <result>
```

```
result: (inj (core <sys-object>)) or,
        (mod (core <sys-object> 0/1 0/1 0/1/2)) or,
        (<function> <arglist> <delay cycles=0>) // general form, the target sub-system is generally
        the first arg or,
        <code> // when the program is in the right side of a red operator lead evaluation
    sys-object, code, arg: explicit definition or reference
    function: reference
```

`sem count`: semaphore count, i.e. number of instances allowed for simultaneous instances, <0 meaning infinite. The `sem count` is decreased by 1 each time an instance of the program is injected. Instantiation results from generating a program from a template defined with variables (an instance is not a duplicate). If at injection time `sem count` is 0, the injection is not performed and a notification occurs in a form similar to a regular process notification:

```
|pro <denied application> <producer> <input_object>.
```

`sem count` is increased by 1 when the program instance dies, or its activation falls below the sys-realm activation threshold. In such cases, and if `sem count>=1`, the `sem count` is decreased by 1 when the activation of the program instance is put back above the threshold. If already 0, notification occurs as described. `sem count` increases or decreases are performed immediately during evaluation.

A program generates a result whenever an object matches the pattern and the guards attached to the result specification are satisfied.

`|pgm` is an anti-program: it generates all of its results if *no* object matches the pattern and the guards attached to the result specification are satisfied.

Notes

As any sys-object, input objects held by reference blocks are controlled by their intensity and activation values. When guards reduce reference blocks, only the objects in the block with an intensity above the sys-realm intensity threshold are reduced. If guards use the reference

block to reduce other fragments, only the programs in the block with an activation above the activation threshold in the sys-realm will attempt the reduction.

Sometimes, the results and their guards are parameterized by the input object. In that case, an anti-program would not be able to generate these results.

EFN Explicit functions.

```
fn_name: efn
    <abstraction>
    ()
        <implementation_0>
        ...
        <implementation_n>
    <realm=nil>
    <sem count=-1>
```

abstraction: spc <fun-pattern> <guards>

fun-pattern: <args>

arg: argument for the function, the first being generally the sub-system exhibiting the function.

implementation: (<guards> <application>) or (<guards> <fragment>).

sem count: as for programs, where “application of the function” replaces “instance of a program”. sem count is decreased by 1 each time an application is injected. If at this time sem count is 0, the injection is not performed and a notification occurs. sem count is increased by 1 when the process resulting from the application finishes (occurrence of a mk.xet), or the application is killed before the process is generated, or the application’s intensity gets below the sys-realm intensity threshold. In such cases, and if sem count>=1, sem count is decreased by 1 when the intensity of an application is put back above the threshold. If already 0, notification occurs.

Variables are valuated from abstraction to implementation. The semantics of explicit functions is the same as usually found in functional language, i.e. explicit functions encode the semantics of an abstraction.

|efn is undefined.

Notes

Functions can be called only as productions in programs, by functions or by models.

Reduction graphs are encoded as graphs of functions both explicit and implicit: the implementation parts reference reductions to be performed subsequently in time while the detection parts (of implicit functions ifn) define pattern detection to be performed priory.

IFN Implicit functions.

```
fn_name: ifn
    <abstraction>
    ()
        <detection_0>
        ...
        <detection_n>
    <sem count=-1>
```

abstraction: spc <fun-pattern> <guards>

fun-pattern: <args>

arg: argument for the function, the first being generally the sub-system exhibiting the function.

detection: (spc <pattern> <guards>) or <application> or <fragment>.

sem count: as for programs, where “application of the function” replaces “instance of a program”. sem count is decreased by 1 each time an application is injected. If at this time sem count is 0, the injection is not performed and a notification occurs. sem count is increased by 1 when the process resulting from the application finishes (occurrence of a mk.xet), or the application is killed before the process is generated, or the application’s intensity gets below the sys-realm intensity threshold. In such cases, and if sem count>=1, sem count is decreased by 1 when the intensity of an application is put back above the threshold. If already 0, notification occurs.

Variables are valuated from detection to abstraction. The semantics of implicit functions is the encoding in an abstraction a pattern over correlated objects in the system. Implicit functions are coupled pattern detectors, i.e. they build abstractions from the detection of

several objects matching a set of patterns. As many abstractions are produced as there are tuples of objects matching the patterns. The detection parts shall resolve eventually in either a pattern specification or an application of an implicit function. It is thus allowed to encode a detection as an application of an explicit function that is implemented in the end at least by one application of an implicit function. The abstraction can feature variable definitions and these can be used to parameterize the detectors.

`|ifn` is undefined.

Notes

Functions can be called only as productions in programs, by functions or by models.

Reduction graphs are encoded as graphs of functions both explicit and implicit: the implementation parts (of explicit functions `efn`) reference reductions to be performed subsequently in time while the detection parts define pattern detection to be performed priory.

APP Application.

`<reference> <args> <delay cycles=0>`

`reference`: to a function, a forward model, an inverse model, or a pregnance.

`arg`: any code or code fragment, the first being generally a sub-system to execute the function, model or pregnance.

`delay cycles`: an integer value n. Indicates if the corresponding process object shall be generated immediately (n=0 i.e. will appear at the same cycle as the application object) or not (n>0, would appear n cycles later). If delayed, the process will be generated if and only if the application is intense enough at this point in time.

Notes

The `app` opcode is implicit in the definition of an application.

Process objects are generated automatically (with a potential delay) upon injection of application objects.

Examples

ex1:

```
move: efn
    spc (self :translation :duration) nil)
    ()
        nil (set_motors (dev.locomotion (/ translation duration) duration)) // application
```

ex2:

```
move_object: efn
    spc (self :object :destination :t) nil)
    ()
        nil (grab (self object)) // attach object to hand
        nil
            inj (core (sys
                      pgm
                          spc (sys grab_pro: (pro (grab (self object)) ::) :mrk1 ::) // when
attached
                              ()
                                  red mrk1
                                      pgm
                                          spc (sys (mk.xet grab_pro ::) : : : :t1) ((= t1 _rdc))
                                          (nil grab_pro)
                          ()
                              nil (move_hand (ent.x destination (- t (+ duration_grab
duration_release)))) // move hand towards destination
                      0 1 run_once))
        nil
            inj (core (sys
                      pgm
                          spc (sys move_hand_pro: (pro (move_hand (self object)) ::) :mrk2 ::) //
when destination reached
                              ()
                                  red mrk2
                                      pgm
                                          spc (sys (mk.xet move_hand_pro ::) : : : :t2) ((= t2
_rdc))
                                          (nil move_hand_pro)
                          ()
                              nil (release_grasp (ent.x)) // release object
                      0 1 run_once))
```

ex3:

```
object_moved: ifn
    spc (ent.x start from to) nil // can also feature input variables
    ()
        spc (sys ent.:x :mrk ::)
        ()
            red mrk
                pgm
                    spc (sys (mk.last_position ent.x :from :start) ::) nil
                    (nil start)
            red mrk
                pgm
                    spc (sys (mk.at ent.x :to) : : : : :t1)((< start t1))
                    (nil t1)
```

An application of `object_moved` by a program `pgm1` upon matching an object `in` generates:

`p: (pro (object_moved (entity start from to) pgm1 in))` and `(mk.xet p duration_rdc duration_ms) // NB: duration_ms >= t1`

Works with:

```
position_updater: pgm
    spc (sys o: (pro (object_moved (ent.:x : :from :to)) ::) :mrk :int :act :res)
        (red mrk
             pgm (spc (sys (mk.xet o :t ) ::) nil) ((nil t))
        )
    (nil (inj (core (sys (mk.last_position ent.x to t) int act res))))
```

ex4:

```
object_moved_at_night: ifn
    spc (ent.x t from to) ((= t t2) (< :lumen 0.1))
    ()
        object_moved (ent.:x :t :from :to)
        spc (sys (mk.ambient_luminosity :lumen) : : : :t2)
```

Invocation:

```
object_moved_at_night: ifn
    spc (ent.x t from to) ((is_nighttime (clk t)) // is_nighttime: function added to clk, user-
defined
    (object_moved (ent.:x :t :from :to))
```

or

```
object_moved_at_night: ifn (spc (ent.x t from to) nil) ((object_moved (ent.:x :t :from :to)))
```

and

```
pgm
    spc (sys (tck :t ::) ::) ((is_nighttime t))
    (nil (object_moved_at_night (::)))
```

FMD Forward model.

```
model_name: fmd
    <abstraction>
    <application-spc>
    ()
        <predictor0>
        ...
        <predictorn>
    <sem count=-1>
```

`abstraction`: pattern specification.

`application-spc`: `spc <application-pattern> <guards>`.

predictor: `<guards> <program>`.

`sem count`: as for functions, the application being the execution of the model.

A forward model defines the outcome of executing actions. An outcome is the production of objects (generally marked as predictions – marker `prd`). Actions are specified as patterns of function applications.

A forward model is executed in the same fashion a function is, i.e. using an application object.

`|fmd` is undefined.

Notes

The anticipated time of the occurrence of the predicted sys-object is the injection time of the predicted object, i.e. in the future.

To execute a forward model: `(<model> <args> <delay cycles=0>)`, just like a standard function application.

Examples

ex1:

```
fmd1: fmd
    spc (self :t :speed :duration) nil
    spc (motors (dev.locomotion t speed duration)) nil
    ()
        nil
            pgm
                spc (sys self :mrk :int :act :res t)
           (red mrk (pgm (spc z: (sys (mk.at_location self :y) ::) nil) ((nil z))))
                ()
                    nil (inj (core m: (sys (mk.at_location self (+ y (* speed duration))) int act
res (+ t duration))))
           nil (inj (core (sys (mk.prd m 1) int act res)))
```

NB: this model deals with locations as linear distances. Dealing with heading would require vectors as a user-defined object (along with related specific operators and upgrades of existing ones).

Execution:

```
fmd1 (self _now 10 150)
```

ex2:

```
fmd2: fmd
    spc (self ent.:x :destination :t) nil
    spc (move_hand (self ent.x destination t) nil // assumes the hand holds ent.x
    ()
        nil
            pgm (spc y: (sys ent.x :mrk :int :act :res :) nil)
                ()
                    nil (inj (core m: (sys (mk.at_location ent.x destination) int act res t)))
                    nil (inj (core (sys (mk.prd m 1) int act res)))
)
```

Execution:

```
predictor: pgm (spc (sys (pro (move_hand (self ent.:object :dest :t)) ::) ::) nil) ((nil (fmd2 (self
ent.object dest t))))
```

with

stimulator: `pgm (spc (sys (pro (grab (self :)) ::) ::) nil) ((nil (mod (core predictor 1 0 1)))) // grab assumed to be atomic here (pro with no duration)`

inhibitor: `pgm (spc (sys (pro (release (self :)) ::) ::) nil) ((nil (mod (core predictor 0 0 1)))) // idem for release (would have to check for xet marker)`

IMD Inverse model.

```
model_name: imd
    <abstraction>
    <target-state-spc>
    ()
        <implementer0>
        ...
        <implementern>
    <sem count=-1>
```

`abstraction`: pattern specification.

`target-state-spc`: `spc <pgm-pattern> <guards>`.

`implementer`: `<guards> <program>`.

`sem count`: as for functions, the application being the execution of the model.

An inverse model defines the actions to be performed to reach a state. Actions are specified as applications, states as patterns.

As forward models, inverse models are executed using an application object.

`|imd` is an anti-inverse model. It specifies actions to perform so that *no* object will match the target state specification.

Notes

To execute an inverse model: `(<model> <args> <delay cycles=0>)`, just like a standard application.

To compute an inverse model: `inv (core <state> <timeout>)`

`state`: can contain variables (partially specified) or not (fully specified).

Example

```
imd1: imd
    spc (self :Ltarget :ttarget) nil)
    spc (sys (mk.at_location self Ltarget) : : : : t) ((<= t ttarget ))
    ()
        nil (pgm (spc (sys (mk.at_location Lcurrent) : : : : :tcurrent) ((<= tcurrent ttarget))
         (nil (set_motors (dev.locomotion tcurrent (/ (- Ltarget Lcurrent) dt: (- ttarget tcurrent))
dt))
```

Execution:

```
imd1 (self some_location (+ _now 1000))
```

This inverse model can be hand-crafted, but can also be produced by explicit inverse resolution, given adequate forward models like `fmd1` as defined in the previous example:

```
inv (core (spc (sys (mk.at_location :L) : : : :t) nil) 400)
```

If `fmd1` constrains its application specification with `(<= speed speed_limit)`, then the inverse resolution call above yields:

```
imd
    spc (self :Ltarget :ttarget) nil
    spc (sys ( mk.at_location self Ltarget) : : : : ttarget) ((<= t ttarget))
    ()
        nil (pgm (spc (sys (mk.at_location Lcurrent) : : : : :tcurrent)
                 ()
                    <= tcurrent ttarget
                    <= tcurrent (- ttarget dt: (/ (- Ltarget Lcurrent ) speed_limit))
                 )
         (nil (set_motors (dev.locomotion tcurrent speed_limit dt)))
```

NB: constraints have to be expressed as reachable values (ex: using `<=` and not `<`)

PRO Process.

`process_name: pro <application> <spawner> <input object> <realm=nil>`

`arg`: any code or code fragment.

`application`: reference to an application.

`spawner`: the program having generated the application (automatic generation of process) or the process itself (programmatic generation).

`input object`: the object that triggered the generation of the application or the process.

`realm`: realm associated to the process, if any.

A process encodes the fact that an application (function, model or pregnance) is being performed. A process can be associated to a realm.

`|pro` is an anti-process. It is automatically generated when a program (the producer) has attempted to generate an application or to inject a program for which *no* resources are available (`sem count=0`).

Notes

Processes are generated automatically following an application, or explicitly by programs. In the latter case, the last two arguments are valuated automatically.

When a process finishes, a completion marker `mk.xet` is created automatically, indicating the process duration. Process completion is reached if:

- the application is a function call and the call returns, or
- the application is the execution of a model and the model has generated its outputs, or
- the application is the instantiation of a pregnance.

The process' starting time is its injection time.

`pro (del <sys-object>) core nil` is a process notifying imminent garbage collection. It is automatically generated when the resilience of the sys-object falls down to 0. This notification gives a chance to save doomed objects. If its resilience is left to 0, the sys-object

will be killed at the end of the current cycle. In any case the process will be killed at the end of the current cycle.

Injection of a process object *does not* trigger the automatic construction of an associated realm.

TCK Clock tick.

`tck <rdc> <t> <avg> <foreseen>`

`rdc`: reduction cycle count.

`t`: time (ms).

`avg`: average duration (ms) of one cycle.

`foreseen`: expected duration (ms) of the cycle to be run.

Notes

Clock ticks are injected at each reduction cycle by `clk`. They have a resilience of 1.

RLM Realms.

realm_name: `rlm <reference> <thr_act_pos> <thr_act_neg> <thr_int_pos> <thr_int_neg> <delay>` *<policy=0> <rng_act_pos=nil> <rng_act_neg=nil> <rng_int_pos=nil> <rng_int_neg=nil>*

`reference`: `<pregnance>` or `<saliency>` or `<process>` or `nil`.

`<thr_act_pos>`, `<thr_act_neg>`, `<thr_int_pos>`, `<thr_int_neg>`: thresholds for intensity and control values, defined for positive and negative contributions.

`<rng_act_pos>`, `<rng_act_neg>`, `<rng_int_pos>`, `<rng_int_neg>`: ranges of the control values defined in the realm, computed from their respective thresholds. They are read only and set initially to `nil` when the realm is created.

`delay`: the period (in reduction cycles) at which the ranges are updated.

`policy`: 0 add the object control value defined in a realm to the object control value in the embedding realm,

1 if realm threshold>=object control value, add the object control value to the object control value in the embedding realm, else add nothing,

2 as 1 but if realm threshold<object control value hide the object from any other realm (incl. sys-realm), i.e. final control value=0.

Realms are objects representing the dimensions of state spaces. They are associated with arbitrary reference processes, (of type `pro`, `sln` or `pre`). Objects are projected onto realms by building projection markers (`mk.pji`, `mk.pja`) that hold contribution values, akin either to intensity control values or to activation control values. These values represent the contribution (positive or negative) of the projected object to the achievement of the reference process, either as input object (intensity value) or programs (activation value). A positive contribution is the depth – expressed in time units - of an object in a rewrite graph that actually supported the reference process. For example, a graph that produced at time *t* an object matching at time *t'* the specification hold by a pregnance would be rated as contributing positively with a value, i.e. a projection, of *t'-t*. A negative contribution is the depth of an object in a graph that hampered the reference process. Another example – this time of a negatively contributing object - would be a graph that lead to deactivating the production of an object matching the specification hold by a pregnance. Projections are valuated in $\mathbb{R}^*$, 0 being the termination of the process. Control values are computed as the inverse of these projections.

As objects, realms can be projected on other realms, in other words, realms can define sub-spaces of the state space. There exist a pre-defined realm, called sys-realm (noted `srm`) that holds no reference process. It is the root of all spaces, i.e. the global state space.

An object projected on several realms is associated by as many local contribution values. These values are automatically combined to form final control values that will be attached to the object in question in the sys-realm. Local contribution values are filtered by two realm-defined thresholds (respectively for input data and programs) and there are currently three methods for applying thresholds and combining local values into a final one (defined by `policy`).

A realm keeps track of the ranges of local contribution values (`rng_int_xxx` and `rng_act_xxx` both defined for positive and negative contributions); these are updated at variable rates, specified by `delay`. This is provided as a convenience for detecting new projections holding

“out of range” control values, i.e. salient projections with regards to existing distributions of control values.

As a sys-object, a realm is also controlled by its final intensity value. When a realm is not activated, the projections it holds are not taken into account for computing the final control values of objects in the system.

`|rlm` is undefined.

Notes

Not all objects have to be projected in a realm.

Realms are constructed automatically whenever pregnances or saliency objects are injected. This is not the case for regular processes (`pro`), for which the decision is left to the application.

Realms are usually populated by an application of the function `ctx`. However, modifying, adding and removing projections programmatically is allowed.

PRE Pregnance.

```
pregnance_name: pre
    <abstraction>
    <expectation>
    <realm=nil>
    <sem count=-1>
```

`abstraction: pattern specification; similar to the abstraction in explicit functions.`

`expectation`: `spc <pattern> <guards>`, as a function detector.

`realm:` the realm – if any – associated to the pregnance.

`sem count`: as for functions, the application being the instantiation of the pregnance.

Pregnances represent internal drives, and by extension, goals. They define an abstraction that specifies a state of the system, e.g. the occurrence of an object, the occurrence of tuples of objects, etc. Pregnances can be reference processes for realms.

`|pre` is an anti-pregnance: it expresses the drive that *no* object exists that matches the expectation.

Notes

To instantiate a pregnance: `(<pregnance> <args> <delay cycles=0>)`, like a standard application. This generates a process, and a completion marker will also be generated when the pregnance’s intensity gets lower than the sys-realm intensity threshold, or if the pregnance gets killed.

`|pre ... <|expectation> ...` is equivalent to `pre ... <expectation> ...` but not to `|pre ... <expectation> ...` nor to `pre ... <|expectation> ...`

The injection of a pregnance triggers the automatic building of a corresponding realm. The contribution of objects to the satisfaction of pregnances can be computed by the executive upon a call to the function `ctx`.

Example

```
be_at_location_100mAhead_tomorrow: pre // 100mAhead being a reference to an existing location
    spc nil nil
    spc (sys self :mrk ::)
        (red mrk
             pgm (spc (z: sys (mk.at_location self L) : : : : :t) ((< t (+ _now 86400000)))) //
rough deadline: now+1day(in ms)
                 (nil z)
        )
```

Works in conjunction with:

```
inj (core (sys (be_at_location_100mAhead_tomorrow nil) 1 0 1000))
```

Variant:

```
be_at_location_L_at_time_T: pre // L and T are now variables
    spc (self :L :T) nil
    spc (sys self :mrk : : : T)
        (red mrk
             pgm (spc (z: sys (mk.at_location self :L) : : : : T) nil) ((nil z))
        )
```

Works in conjunction with:

```
inj (core (sys (be_at_location_L_at_time_T (self 100mAhead (+ _now 86400000))) 1 0 1000))
```

Upon injection of `be_at_location_L_at_time_T`, a realm, say `rlm1`, is created by the executive and the `<realm>` slot in `be_at_location_L_at_time_T` now contains a reference to `rlm1`.

SLN Saliency.

saliency_name: `sln <abstraction>` *`<realm=nil>`*

`abstraction`: pattern.

`realm`: the realm – if any – associated to the saliency.

Local representation of an abstract (or geometric) saliency. Saliency holds goal semantics, i.e. a saliency object would *attempt* to observe more or less specific facts, specified by the abstraction. A saliency object is thus used as pregnances to define dimensions of the state space, and any process can be projected on saliency objects to express their contribution to the occurrence of objects matching the abstraction.

`|sln` is an anti-saliency, that is, the saliency of *no* object matching the abstraction.

Notes

The injection of a saliency object triggers the automatic construction of an associated realm.

A2.4 Sub-system functions

INJ Inject an object.

`inj (core <sys-object>)`

`sys-object`: any sys-object.

Injects the sys-object in the sys-realm. Returns the projections (`mk.pja` and `mk.pji`) of the object on the sys-realm.

Notes

No application is injected, but a process is generated, tagged by an execution time marker (`xet`) if the injection was successful.

The sys-object can have an injection time set in the future.

MOD Modify a control value.

`mod (core <object> <operation> <selection> <target>` *`<delay=0>`*`)`

`object`: realm, projection or sys-object (in that case resilience must be the target).

`operation`: increase (1), decrease (0).

`selection`: base control value (0), control value (1): only applies to int, act, res, thr_int, thr_act and delay.

`target`: intensity (0), activation (1), resilience (2), threshold intensity positive (3), threshold intensity negative (4), threshold activation positive (5), threshold activation negative (6), delay (7), policy (8), sem count (9) (when defined).

`delay`: in reduction count. Postpones the performance of `mod` in the future.

Modifies one control value of the target with an amount specified by the base control value. If the target is a base control value, then the modification amounts to 1.

Notes

No application is injected, but a process is generated.

SET Set a control value.

`set (core <object> <selection> <target> <value>` *`<delay=0>`*`)`

`object`: realm, projection or sys-object (in that case resilience must be the target).

`selection`: base control value (0), control value (1): only applies to int, act, res, thr_int, thr_act and delay.

`target`: intensity (0), activation (1), resilience (2), threshold intensity positive (3), threshold intensity negative (4), threshold activation positive (5), threshold activation negative (6), delay (7), policy (8), sem count (9) (when defined).

`value`: any number.

`delay`: as for `mod`.

Sets one control value of the target.

Notes

No application injected, but a process is generated.

GET Get a control value.

```
get (core <object> <selection> <target>)
```

`object`: realm, projection or sys-object (in that case resilience must be the target).

`selection`: base control value (0), control value (1): only applies to int, act, res, thr_int, thr_act and delay.

`target`: intensity (0), activation (1), resilience (2), threshold intensity positive (3), threshold intensity negative (4), threshold activation positive (5), threshold activation negative (6), delay (7), policy (8), sem count (9) (when defined).

Returns one control value of the target immediately upon evaluation.

Notes

No application is injected, no process is generated.

INV Compute inverse models.

```
inv (core <specification> <timeout>)
```

`specification`: `spc <fragment> <guards>`.

`timeout`: in ms. If set to -1, `inv` will execute until the next reduction cycle.

Attempts to compute the inverse model that, when executed would inject an object matching the specification. It is a backward evaluation process that uses equational rewrite rules (`erw`) to infer possible reduction graphs that would produce objects matching `specification` from existing objects (e.g. predictions in the forward models, existing reduction graphs, etc.). If successful, an inverse model is injected, otherwise (the function times out) the reduction graph candidates found so far cannot be initiated by any object in the system. Pregnances are then created and injected, expressing the need of objects able to trigger or activate the reduction graph candidates, or the need to deactivate or suppress objects that deactivate possible candidates.

`inv` performs only on a sub-set of the system. A sub-set is identified by a realm, associated to the process of executing a particular application of `inv`. This realm is created automatically upon the application of `inv`, meaning that by default, the entire system will be investigated. Subsequent calls to `inv` will make use of projections determined either by `ctx` or by programs. In order to restrict the search space beforehand, the realm must be created programmatically, and this also the case for the initial projections.

Notes

No application is injected, but a process is generated.

CTX Compute contexts.

```
ctx (core <process> <timeout>)
```

`process`: `(<pregnance> <args>)`, `(pro ...)` or `(sln <args>)`.

`timeout`: in ms. If set to -1, `ctx` will execute until the next reduction cycle.

Populates the realm associated to a process with projections of sys-objects (markers `mk.pji`, `mk.pja`). The function returns either when each object has been evaluated (and possibly projected) or when it times out. Object evaluation is performed by starting from the end of rewrite graphs having led to the termination of the process (marker `mk.xet` with no killer) and backtracking towards including older objects, i.e. located earlier in the graph. Positive contributions are also found in reduction graphs that contributed negatively to anti-processes or process abortion (`mk.xet` with a killer). Negative contributions are evaluated in parallel, finding other reduction graphs that attempted to inhibit participants in positive graphs, or finding positive contributions to anti-processes or process abortion.

Notes

No application is injected, but a process is generated.

BTT Get the boot time.

```
btt (clk)
```

Returns the boot time (ms).

Notes

No notification (i.e. no application nor process will be generated).

NOW Get the current time since boot time.

```
now (clk)
```

Returns the current time (ms).

Notes

No notification.

RDC Get the reduction count since boot time.

```
rdc (clk)
```

Returns the current reduction cycle count.

Notes

No notification.

RDP Set the reduction depth.

```
rdp (cfab <depth>)
```

Sets the reduction depth, i.e. the number of reduction cycles to perform before admitting inputs from the sub-systems, and updating the memory across the computing nodes.

Notes

No application is injected, but a process is generated.

RDQ Set the reduction quanta.

```
rdq (cfab <quanta>)
```

Sets the reduction quanta for the sub-system that issued the call.

Notes

No application is injected, but a process is generated.

LDR Code loader.

```
ldr (<spv> <code region> <in/out>)
```

`spv`: a reference to a supervision sub-system.

`code region`: an integer identifying a region in a source file.

`in/out`: 1 (load) or 0 (unload).

(Un)loads code regions in a supervision sub-system. These regions are defined in source code files.

Notes

No application is injected, but a process is generated.

If an object t is member of r0 and r1 and only r0 is unloaded, t will not.

RCT Reduction control.

```
rct (<sub-system> <function>)
```

`sub-system`: a reference to core or to a supervision sub-system.

`function`: 0 (run), 1 (stop), 2 (pause), 3(resume).

Controls the reduction performed by a supervision sub-system.

Notes

No application is injected, but a process is generated.

SWP Swap.

```
swp (<sub-system> <function> <id> <block>)
```

sub-system: a reference to core or to a supervision sub-system.

function: 0 (disk to ram), 1 (ram to disk).

id: integer identifying a file (generally a symbolic link).

block: contains references to objects to be swapped out (only when function=1) if nil, the entire CORE block will be swapped out, leaving only srm and self.

Swaps code to/from disk.

When function=0, the block is irrelevant. The objects are loaded from disk and placed directly in ram.

When function=1, the identifier is optional. If not provided, the file identifier is generated by the sub-system.

swp returns the file identifier or nil in case of trouble.

Notes

No application is injected, but a process is generated.

Code is stored on disk as it is in CORE blocks.

MIN Min monitor.

```
min (core <application/process> <selection> <control> <on/off=nil>)
```

application/process: a pattern specifying any application (function, model, etc) or process.

selection: a pattern identifying the application output value to be monitored (there can be several).

control: 0/1; constrains the search space to objects with an intensity value above the sys-realm threshold (1) or performs regardless of these values (0).

on/off: 0/1 to start/stop.

Starts/stops the monitoring of the minimal value of the specified application. When the minimal value changes a marker (mk.min) is attached to the application.

min searches the system for applications with a positive resilience. To include objects before they get garbage-collected, set start to 1; otherwise set start to nil.

Note

Monitor semantics.

No application is injected, but a process is generated.

MAX Max monitor.

```
max (core <application/process> <selection> <control> <on/off=nil>)
```

application/process: a pattern specifying any application (function, model, etc) or process.

selection: a pattern identifying the application output value to be monitored (there can be several).

control: 0/1; constrains the search space to objects with an intensity value above the sys-realm threshold (1) or performs regardless of these values (0).

on/off: 0/1 to start/stop

Starts/stops the monitoring of the maximal value of the specified application. When the maximal value changes a marker (mk.max) is attached to the application.

max searches the system for applications with a positive resilience. To include objects before they get garbage-collected, set start to 1; otherwise set start to nil.

Note

Monitor semantics.

No application is injected, but a process is generated.

AVG Average monitor.

`avg (core <application/process> <selection> <control> <on/off=nil>)`

`application/process`: a pattern specifying any application (function, model, etc) or process.

`selection`: a pattern identifying the application output value to be monitored (there can be several).

`control`: 0/1; constrains the search space to objects with an intensity value above the sys-realm threshold (1) or performs regardless of these values (0).

`on/off`: 0/1 to start/stop

Starts/stops the monitoring of the average value of the specified application. When the average value changes a marker (`mk.avg`) is attached to the application.

`avg` searches the system for applications with a positive resilience. To include objects before they get garbage-collected, set `start` to 1; otherwise set `start` to `nil`.

Note

Monitor semantics.

No application is injected, but a process is generated.

RND Random number generator.

`rnd`

Convenience function. Returns a number in [0,1].

Note

No application is injected; no process is generated.

A2.5 Markers

Markers are objects used to attach descriptions on existing objects. They are instantiated from existing marker classes, defining the ontological concepts of a given system.

PJA, PJI Projection markers.

marker_name: `mk.pj<x> <sys-object> <realm> <value>`

`x`: in `{a,i}`.

`realm`: reference to a realm.

`value`: contribution (akin to activation for `mk.pja`, or intensity for `mk.pji`) for the sys-object projected on the realm.

A projection encodes the contribution of an object to the achievement of the process the realm is associated to. When the object is a program the contribution is used to compute the activation value, whereas when the object has been an input data for reductions, the contribution is used to compute the intensity value. Contributions are numbers in $\mathbb{R}^*$: they represent the depth – in time units - of a given object in a rewrite graph measured from the time of the contribution to the process: termination, intensification, activation, etc. Final control values are computed as a combination of the inverse of the contributions (see realm policy).

Notes

Projections on the sys-realm are the result returned by the `inj` function.

To inject projections on p-realms programmatically is allowed.

INS Instance marker.

marker_name: `mk.ins <sys-object> <category>`

`sys-object`: any sys-object.

`category`: any sys-object.

Indicates that the sys-object is an instance of another sys-object, acting as a category.

Notes

This instance relationship is not structural (like a template being instantiated or pattern matched), but purely axiomatic.

HYP Hypothesis marker.

marker_name: `mk.hyp <sys-object> <actuality>`

`sys-object`: any sys-object.

`actuality`: in [0,1], 0: indicates a non existent fact, 1 a true fact.

Indicates that the sys-object is an hypothesis.

Notes

Shall be generated by programs, functions, or models to mark their productions when they result from processing hypothesis.

SIM Simulation marker.

marker_name: `mk.sim <application or program>`

`application or program:` reference to any application or program.

Indicates that an application is a simulation or a program runs in simulation mode. If the application or the program execution is resolved to an actual call of a sub-system function, it will not be performed.

Notes

Any production resulting from the application or the program are automatically marked as simulations.

Simulated objects are different from hypothetic objects. The former are objects certain to be obtained if the application/program ran in real (non-simulated) mode, whereas hypothesis are uncertain. If an object is marked as both a simulation and an hypothesis it means that it is an hypothesis that would be generated if the producer ran in real mode.

PRD Prediction marker.

marker_name: `mk.prd <sys-object> <confidence>`

`sys-object`: any sys-object.

`confidence`: in [0,1], 0: indicates an impossibility, 1 a certainty.

Indicates that the occurrence of the sys-object is a prediction with regards to a confidence value.

Notes

Generated by forward models, programs or functions.

XET Execution time of a process.

marker_name: `mk.xet <process> <killer> <duration rdc> <duration ms>`

`process`: reference to any process.

`duration`: of the process both in reduction count and ms.

`killer:` reference to the object that aborted the process, if any.

Indicates the termination of a process.

Notes

Automatically generated when the corresponding application terminates.

The process termination time is given by the injection time of the `xet` marker.

`mk.xet` carries the time a particular reduction took; to be used in comparison to the total time of a reduction cycle, given by `tck`.

MIN, MAX, AVG Minimal/maximal/average application values.

marker_name: `mk.<x> <min/max/avg application>` <*application/process*> `<value>`

`x:` in `{min,max,avg}`.

`min/max/avg application`: the application of a `min/max/avg` function.

`application/process:` the actual application (or process) that produced the value – only for `min` and `max`. For average, the application/process is set to `nil`.

`value`: the minimal/maximal value of the application.

Tags an application object (`application`) that produced a minimal maximal or average value. These values are monitored by the eponym functions.

Notes

Automatically generated by `min`/`max/avg` functions.