# Challenging Questions for the Rationals of Non-Classical Programming Languages

Olivier Michel[1], Jean-Pierre Banâtre[2], Pascal Fradet[3] and Jean-Louis Giavitto[1]

[1] LaMI – CNRS – Université d'Évry – Genopole, France[*]
[2] IRISA – Université de Rennes 1, France[†]
[3] INRIA Rhône-Alpes, Grenoble, France[‡]

In this note, we question the rationals behind the design of unconventional programming languages. Our questions are classified in four categories: the sources of inspiration for new computational models, the process of developing a program, the forms and the theories needed to write and understand non-classical programs and finally the new computing media and the new applications that drive the development of new programming languages.

## 1 Metaphors for Computations

Programming paradigms, or their concrete instantiations in programming languages, are not coming "out of the blue". They are inspired either by the peculiarities of a computer or by a metaphor of what a computation should be. A few examples: the typewriter for the Turing machine, desk, scissor and trash can for user-interfaces, classification and ontology for the object based languages, building and architecture for design patterns, meta-mathematical theories (e.g. $\lambda$-calculus) for functional programming. Considering the programming languages history, it seems that the metaphors that are really working are mainly based on artifacts or on the notions and concepts that structure a domain of abstract activities (office, mathematics).

We are now experiencing a renewed period of proposals based on "*natural* metaphors": artificial chemistry, DNA computing, quantum computing, P systems, PPSN (parallel problem solving from nature: simulated annealing, evolutionary algorithms, etc.), cell and tissue computing… This is not to say that the metaphors of the biological and physical world were absent until now. On the contrary, formal neurons and cellular automata, both inspired by biological notions, have been elaborated from the very origin of computer science. But this opposition between the relatively few impacts of natural metaphors in programming language compared to the large widespread of metaphors of other human specific activities, asks from the following interrogations:

- What do we expect from the natural metaphors that we do not have with the metaphors of human activities? To answer which needs, to support which applications, to answer which failures?

- What are the links between Physics and Computation? Physics obviously determines the phenomena that can be used for computing (the hardware). However, on what

[*]LaMI umr 8042 CNRS – Université d'Évry, Tour Évry-2, 523 place des terrasses de l'agora, 91000 Évry, France. `[michel, giavitto]@lami.univ-evry.fr`

[†]IRISA Campus de Beaulieu, 35042 Rennes Cedex, France. `Jean-Pierre.Banatre@irisa.fr`

[‡]INRIA Rhône-Alpes, 655 avenue de l'Europe, Montbonnot, 38334 Saint Ismier Cedex, France. `pascal.fradet@inria.fr`

extend can it be a source of inspirations for programming? For instance, what are the impacts on programming of Feynman's lectures on the physics of the computation? What lessons have we learned from the "analog computation" developed during the '50s and the '60s?

- What are the links between Biology and Computation? Biology is obviously a source of inspirations for new computational models. And the computer scientists are desesperately looking for design principles to achieve systems with properties usually attributed to life: self-sustaining systems, self-healing systems, self-organizing systems, autonomous systems, etc. However, are we sure to agree on the meaning of these characteristics? For example, these properties are often exhibited at a collective level at a large scale and on the long time, not at the level of an individual: a specie, robust against the variations of its environment, does not mean that the individuals adapt easily to these variations.

- Do we have exhausted the metaphor of human activities (engineering, liberal art, economics, math, literature, philosophy, etc.)? For instance, logic and meta-mathematics are tightly coupled with computer science. What about geometry or topology? (the geometrization of physics since the end of the nineteenth century is a major trend but it does not seem to appear in computation).

- Is the physical world a right source of inspirations? In other words, are the relationships between physical objects a good framework to conceptualize the relationships between immaterial objects like softwares or computations? For example, *synchronous languages* make the assumptions that the reaction to events are instantaneous. Despite the apparent violation of physical laws, this model is very successful to reason and implement real-time applications.

## 2 Programming in the Small and Programming in the Large

**Programming in the Small.** The slogan:

$$\textit{program} = \textit{data-structures} + \textit{algorithms}$$

has shaped our approach of what a program is.

- Is this manifesto still relevant in front of the new programming paradigms (and the new problems and applications)?

- What are the new data-structures offered by the chemical, the tissue and other computing paradigms?

- Are there new algorithms or only a speed-up of existing mechanisms?

Control structure are the means by which we organize the set of computations that must be done to achieve a given task. Organizing natural computations seems very difficult: thinks on how to really implement sequentiality in a chemical computation (e.g. how to start a given chemical reaction in a test tube only whenever the equilibrium of another one has been reached?) . This issue is perhaps related to Landin's splitting of a programming language into two independent parts: (a) the part devoted to the data and their primitive operations supported by the language, and (b) the part devoted to the expression of the functional relations amongst them and the way of expressing things in terms of other things (independently of the precise nature of these things). An example of the latter is the notion of identifier and the rule about the contexts in which a name is introduced, defined, declared or used. The appropriate choice of data and primitive gives an "API" or a "problem-oriented", "domain specific", "dedicated" language. A good choice of the features in the second part can make a language flexible, concise, expressive, adaptable, reusable, general. So,

- What are the new control structures?
- Are the new programming paradigms concentrating only on dedicated and specialized data-structures and operations well fitted to optimize some costly specialized task; or is there also some emergence of *new ways of expressing things in term of other things*?

**Programming in the Large.** The research on chemical computing, biological computation, quantum computing, . . . , mainly focus on the complexity achieved for algorithmic tasks (sorting, prime factorization, etc.). These studies illustrate only the "programming in the small" task and do not address the problem of the "programming in the large", that is the problems raised by the support of large software architecture, the interconnection of modules, the hiding of information, the capitalization and the reuse of existing code, etc. Programming in the large is certainly one of the major challenges a programming language must face.

Concepts of *modules, packages, functors, classes, objects, mixins, design patterns, framework, middleware, software buses, etc.*, have been developed to face these needs. And, following some opinions, *have failed*[1]:

- Is this failure a consequence of the existing programming language or of our methods of software development?
- Why are the programming paradigms discussed here, more fitted to fight against this fragility and inflexibility?
- Which features help to discover/localize/correct program errors or reliably to live with?

**The Disappearing "Software Life Cycle".** For many reasons, the notion of monolithic, standalone, single author program is vanishing. The classic "separate compilation and linking" model of compiler-based languages is no more adapted to our use of programs. After the use of preprocessing and code generation tools we have invented dynamic linking, templates, multi-stage compilation, aspects weaving, just-in-time compilation, automatic

[1] *Gerald Jay Sussman*, in 1999, has written as a justification of the amorphous computing project: "Computer Science is in deep trouble. Structured design is a failure. Systems, as currently engineered, are brittle and fragile. They cannot be easily adapted to new situations. Small changes in requirements entail large changes in the structure and configuration. Small errors in the programs that prescribe the behavior of the system can lead to large errors in the desired behavior. Indeed, current computational systems are unreasonably dependent on the correctness of the implementation, and they cannot be easily modified to account for errors in the design, errors in the specifications, or the inevitable evolution of the requirements for which the design was commissioned. (Just imagine what happens if you cut a random wire in your computer!) This problem is structural. This is not a complexity problem. It will not be solved by some form of modularity. We need new ideas. We need a new set of engineering principles that can be applied to effectively build flexible, robust, evolvable, and efficient systems."
See also the notes of the debate "Object have failed" organized by R. Gabriel at OOPSLA 2002: `www.dreamsongs.org`.

update, push and pull technologies, deployment, etc. In the same time, our systems must include thousands of disparate components, partial applications, services, sensors, actuators on a variety of hardware, written by many developers around the world (and not always in a cooperative fashion).

- In which ways the new programming paradigms can contribute to these trends?

## 3 The Future of Syntax, Semantic, etc.

**The Future of Syntax.** The question of syntax always causes brutal reactions. There is a large trend to become "syntax independent". For example, standards like XML provide flexible and generic tools to translate a deep representation to various surface expressions. In programming languages, features like overloading, preprocessor, macro, combinators, ..., are also used to tailor the syntax in order to offer to the user an interface close to the standard of the application domain. The Mathematica system is a good example of such achievement. However, the deep representation is exclusively relying on the notion of *terms*.

- Does the handling of new programming paradigms require new syntactic representation? Are there needs for diagrammatic, visual, kinesthetic, ..., representations of a program? Will a program necessarily need to be represented as a tree of symbols?

**Semantics and theoretical models.** The influence of logic in the study of the semantics of programming languages is preeminent (with, perhaps, the notable exception of denotational semantics). However, the new programming models seem to put an emphasis on the notion of *dynamical systems*. So:

- What is "the right" mathematical framework allowing the building and the manipulation of dynamical systems in conformity with the concepts of software architectures?
- Can we expect a cross fertilization between theoretical computer science and control system theory?
- Considering the distributed nature of computer resources and of the applications, can we develop a theory of distributed *dynamical systems without a global time or a global state*?
- Are the new paradigms fitted to the development of a notion of "*approximate*", "*probabilistic*", "*fuzzy*", "*non-deterministic*" *computations*? Can they handle in a better way uncertainty and incomplete information?
- Is it possible to define a useful notion of *open systems* within the new paradigms? What are the mechanisms of openness? its control structures? How to maintain coherence and adequation of open systems?

The destiny of a program is to be run in order to accomplish some task. But in order to be sure that the task will be well accomplished, we have developed several concepts and techniques like: typing, static analysis, abstract interpretation, bisimulation, model checking, test, validation, correctness by construction... These techniques consider the program as an object of study. So:

- Are these techniques applicable to the new programs? For instance, what can be the type of a DNA in a test tube? What can be the "correctness by construction" of an amorphous program? What can be the model-checking of a P systems?
- These techniques share the same approach: establishing efficiently and as much automatically as possible, some assertions about programs. This will undoubtedly imposes some (severe ?) limitations on what kind of assertions we can do. We also do not want our assertions be larger than our programs or more difficult to establish than to develop the program.

  Is there an opportunity for other approaches? For example, instead of ensuring statically and *a priori* the correct execution of a program, we can try to correct it incrementally to finally achieve its prescribed task. This approach is tightly coupled with notions like *evolution, emergence, self-organization, learning...* What new approaches of program correction can be supported by the new paradigms?
- More generally, how can the programmer be helped in creating, understanding, enhancing, debugging, testing and reusing programs in the new paradigms?

## 4 New Applications, New Opportunities

**New Computing Resources.** Our favored programming languages often reflect a sequential dogma: we use a step-by-step modification of a global state. This is also true at the hardware level, even in our parallel machines: we partition the processing element between a very big passive part: the memory and a very fast processing part: the processor. While this dogma was adapted to the early days of computers (it can be implemented with as little as 2250 transistors), it is likely to become obsolete as the numbers of resources increases ($10^9$ transistors by 2007). New developments such as nano-technologies or 3D circuits, or more simply parallel multichips systems can potentially provide thousand times more resources.

- So, can new programming paradigms take profit of all this available computational power? The technological progress focused on quantitative improvements of current hardware architecture and little effort has been spent on investigating alternative computing architecture. The point here is not to change from the silicon medium to another one, but to fully exploit the silicon potential! What can we do with this "ocean of gates"?
- Advances in nanosciences and in biological sciences are being used to drive innovation in the design of novel computing architectures based on biomolecules. The ability of DNA and RNA nucleotides to perform massively parallel computations to solve difficult, NP-hard, computational problems are now recognized and DNA molecules will be utilized to construct two- and three-dimensional physical nanostructures, thus providing the ability to self-assemble physical scaffolds. *However*, we already met such opportunities in the past, for instance with optoelectronics: FFT comes at virtually no cost, switching too, etc. But until now, optoelectronic devices have a little impact on our computations. An explanation can be that the operations provided are too rigid and cannot be integrated easily into a more generic framework to allow ease of use and the generality of the applications.

  So, are the new paradigms generic enough? Can they be integrated into mixed-paradigms languages? Can we harness the computationnal power of the new paradigms within more classical languages? What is the price of mixing them? If they are supported by dedicated new hardware, can we interconnect these hardware and make them cooperate at a little cost?

- Do we have to make a difference between bio-inspired (quantum-inspired, chemistry-inspired, *xxx*-inspired...) programming languages and bio-based (quantum-based, chemistry-based, *xxx*-based...) hardware?
- If *hardware* will evolve towards *bioware*, does the *software* must evolve towards *wetware*?

**Programming Immense Interaction Networks.** An area of explosive growth in computing is that of the World Wide Web. Computing over the WWW provides challenges whose solutions will involve the development of new paradigms. One of the challenges is to ensure global requirements (properties of the network as a whole). This challenge *exactly meets* the challenge raised by the programming of smart materials or biological devices: "how do we obtain coherent behavior from the cooperation of large numbers of unreliable parts that are interconnected in unknown, irregular, and time varying ways ?"[2].

- Is there an unified framework that can be useful to reason generically on the collective behavior at a population level, both at a very large scale (the WWW) and at the small scale (nanodevice)?
- What is missing in our current established algorithmic approach, architecture design and formal methods, so that the issues of tolerance, trust, cooperation, antagonism and control of complex global systems cannot be handled properly?

## *Acknowledgements*

The formulation of these questions have benefited from the numerous interactions that have taken place between the participants of the "Unconventional Programming Paradigms" (UPP04) workshop[3].

[2]Gerald L. Sussman, speaking about the programming of programmable materials.
[3]`http://upp.lami.univ-evry.fr`