

Int. J. of Unconventional Computing, Vol. X, pp. 1–11
Reprints available directly from the publisher
Photocopying permitted by license only

©2006 Old City Publishing, Inc.
Published by license under the OCP Science imprint,
a member of the Old City Publishing Group

Challenging Questions for the Rationale of Non-Classical Programming Languages

OLIVIER MICHEL¹, JEAN-PIERRE BANÂTRE², PASCAL FRADET³
AND JEAN-LOUIS GIAVITTO¹

¹*IBISC – CNRS – Université d'Évry – Genopole, France.*

E-mail: michel@ibisc.univ-evry.fr

E-mail: giavitto@ibisc.univ-evry.fr

²*IRISA – Université de Rennes 1, France.*

E-mail: Jean-Pierre.Banatre@irisa.fr

³*INRIA Rhône-Alpes, Grenoble, France.*

E-mail: pascal.fradet@inria.fr

Received: December 4, 2005. In Final Form: March 20, 2006.

In this position paper, we question the rationale 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.

Keywords: unconventional programming language, computing metaphors, syntax, semantics, program development.

1 INTRODUCTION

In this position paper, we do not take a definite position on non-classical programming languages nor do we address a particular concept or approach. Instead, we ask questions and put forward key issues about the design of future programming languages. The questions we have selected emphasize the issues which should guide the development of new programming languages. The field of unconventional “computing models”, which is devoted to the study of the *complexity* of problems using a predefined set of (more or less exotic) basic operations, is not under focus here.

We postulate that the development of new programming languages is driven by the quest of new *expressive power*. The literature on programming language contains a wealth of informal claims on the relative expressive power of programming languages. However, this very notion remains difficult to formalize: for instance, we cannot compare the set of computable functions that a programming language can represent, because nearly all programming languages are universal. As far we know, there are only a few attempts to formalize this notion of expressiveness, see [13, 20]. These works mainly rely on the idea of translating a language into another, using a limited and predefined form of translation (if any translation is allowed, a universal language can be the target of the translation of any other one). However, these notions fail to explain why object-oriented languages (like C++ or Java) are usually considered as more expressive than their imperative counterpart (like C).

We do not try here to develop a theoretical framework able to formalize this kind of concept. We investigate the programming language design space by other means. We advocate that the expressiveness of a programming language can be informally evaluated by considering four criteria :

- the notion of computation embodied into the language,
- the support of the development process,
- the support for reasoning on programs,
- the applications for which it is well suited.

In the rest of this paper, we will try to discuss these four criteria with respect to the recent development of non-classical (natural) computational paradigms: the sources of inspiration for new computational models (section 2), the process of developing a program (section 3), the forms and the theories needed to write and understand non-classical programs (section 4) and finally the new computing media and the new applications that drive the development of non-classical programming languages (section 5). Examples of non-classical (natural) computational paradigms we have in mind are given by the amorphous computing project [7], the autonomic computing initiative [17] and the development of various bio-inspired and chemical computing approaches [6, 8].

2 METAPHORS FOR COMPUTATIONS

Programming paradigms, or their concrete instantiations in programming languages, do not come “out of the blue”. They are inspired either by the peculiarities of a computer or by a metaphor of what a computation should be. As sources of inspiration, we can cite: 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 theory (λ -calculus) for functional programming. Considering the programming languages history, it seems that the most fruitful metaphors have been based on artifacts, notions and concepts that structure a domain of abstract activities (office, mathematics). For example, logic programming is based on the slogan “computation is deduction”, while functional programming relies on the “computation is function application” manifesto.

We are now experiencing a renewed period of proposals based on “*natural* metaphors”: artificial chemistry [12], DNA computing [4], quantum computing [21], P systems [1], PPSN (parallel problem solving from nature: simulated annealing, evolutionary algorithms, etc.) [3], cell and tissue computing [5]. . . to name a few. 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 and motivated by biological abilities, have been elaborated from the very origin of computer science with names like W. Pitts and W. S. McCulloch (formal neurons, 1943), S. C. Kleene (inspired by the previous for the notion of finite state automata, 1951), J. H. Holland (connectionist model, 1956), J. Von Neumann (cellular automata, 1958), F. Rosenblatt (the perceptron, 1958), etc.

This opposition between the relatively few impacts of natural metaphors in everyday programming language compared to the large widespread of metaphors of other human specific activities, asks the following questions:

- What are the benefits of natural metaphors compared to 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, to what extent can it be a source of inspiration for programming ? For instance, what is the impact on programming of Feynman’s lectures on the physics of the computation [14] ? What lessons have we learned from the “analog computation” developed during the 50’s and the 60’s ?
- What are the links between Biology and Computation ? Biology is obviously a source of inspirations for new computational models. Computer scientists are desperately 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, do we understand and agree on the meaning of these characteristics ? For example, the properties of living organisms are often exhibited at a collective level at a large scale and on the long time, not at the level of an individual: a species, robust against the variations of its environment, does not mean that the individuals adapt easily to these variations.

- Have we 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 (however, see [15]).
- Is the physical world a good source of inspirations? In other words, are the relationships between physical objects a good framework to conceptualize the relationships between immaterial objects like software or computation? For example, *synchronous languages* [10] 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.

3 PROGRAMMING IN THE SMALL AND PROGRAMMING IN THE LARGE

3.1 Programming in the Small

The slogan [23]:

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

has shaped our approach of what a program is.

- Is this manifesto still relevant to the new programming, paradigms, problems and applications?
- What are the new data-structures offered by the chemical, tissue and other computing paradigms?
- May unconventional languages suggest new algorithms or only a speed-up of existing ones?

Control structures 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: think about how to implement sequentiality in 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) [18]. An example of the latter is the notion of identifier and the rule about the contexts in which a name is defined, declared or used. The appropriate choice of data and primitive

yields 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 of non-classical programming languages?
- 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*?

3.2 Programming in the Large

Research on chemical computing, biological computation, quantum computing, etc., mainly focuses on the complexity of small 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 issues 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, components, middleware, software buses, etc.*, have been developed to face these needs. And, following some opinions, *have failed* to produce flexible and robust systems¹:

- Is this “failure” a consequence of the existing programming languages 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?

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.” [22]. See also the notes of the debate “Object have failed” organized by R. Gabriel at OOPSLA 2002: www.dreamsongs.org.

3.3 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 not suitable for very large and heterogeneous systems. After the use of preprocessing and code generation tools, programmers have invented dynamic linking, templates, multi-stage compilation, aspects weaving, just-in-time compilation, automatic 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 can the new programming paradigms contribute to these trends ?

4 THE FUTURE OF SYNTAX, SEMANTICS, ETC.

4.1 The Future of Syntax

The question of syntax always causes intemperate 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*.

- Do new programming paradigms require new syntax such as diagrammatic, visual, kinesthetic, . . . , representations ? Or does a program necessarily need to be represented as a tree of symbols ?

4.2 Semantics and Theoretical Models

The influence of logic in the study of the semantics of programming languages is preeminent. 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 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 applications, can we develop a theory of *distributed dynamical systems without a global time or a global state* ?

- Are the new paradigms suited 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*² within the new paradigms? What are the mechanisms and control structures of openness? How can we maintain coherence and adequate behaviour of open systems?

4.3 Validation and Verification

A program’s destiny is to be executed 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, testing, proofs, validation, correctness by construction... These techniques consider the program as an object of study.

- Are these techniques adaptable to the new paradigms? 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? Is it possible to model-check P systems?

These techniques share the same approach: establishing efficiently and as automatically as possible, some assertions about programs. This will undoubtedly impose some (severe?) limitations on the kind of assertions which can be proved or inferred. Assertions should not to be larger than programs or more difficult to establish than to develop programs.

- Are there opportunities for other approaches? Instead of ensuring statically and *a priori* the correct execution of a program, would not it be possible to modify it incrementally so that it achieves its prescribed task? This approach [2, 16] is tightly coupled with notions like *evolution*, *emergence*, *self-organization*, *learning*... What other approaches of program correction can be supported by the new paradigms?
- More generally, how can the programmer be helped in creating, understanding, proving, enhancing, debugging, testing and reusing programs in the new paradigms?

It would be also very interesting to investigate how very high-level languages can bridge the gap between specification languages and lower-level implementation oriented languages. In this context, we consider as very promising methods which allow to derive programs from specifications in a

²i.e., a system that interacts with an unpredictable environment

systematic way. Such methods can rely on specific calculi and disciplines as proposed by E.W. Dijkstra in [11] or as applied in the Chemical Programming setting [9].

5 NEW APPLICATIONS, NEW OPPORTUNITIES

5.1 New Computing Resources

Most programming languages often reflect a sequential dogma: they modify a global state step-by-step. 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 few very fast processing parts: the processors. 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.

- Can new programming paradigms take profit from 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 had little impact on computation. 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.

- Are the new paradigms generic enough? Can they be integrated into mixed-paradigms languages? Can we harness the computational 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?

- Should we draw a line between bio-inspired (quantum-inspired, chemistry-inspired, *xxx*-inspired...) programming languages and bio-based (quantum-based, chemistry-based, *xxx*-based...) hardware?
- If *hardware* evolves towards *bioware*, should *software* evolve towards *wetware*?

5.2 Programming Immense Interaction Networks

An area of explosive growth in computing is that of the Internet or the World Wide Web. Computing over the Web provides challenges asking for the development of new paradigms. One important challenge is to ensure global 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?”³.

- 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 mobile phone network, the WWW) and at the small scale (nanodevice)?
- What is missing in the current established algorithmic approach, architecture design and formal methods, to handle the issues of tolerance, trust, cooperation, antagonism and control of complex global systems properly?

6 CONCLUDING REMARKS

In this position paper, we have considered the impact of new computing hardware and metaphors (e.g. bio-inspired, DNA, chemical or quantum computing) on programming languages issues. These unconventional point of views trigger new questions on basic notions such as data structures, algorithms, syntax and semantics and lead up to reconsider the software development cycle, the verification, reasoning and implementation of programs. Clearly, non-classical programming languages are becoming an ebullient area of research. We believe that the most interesting developments are yet to come.

The formulation of these questions have benefited from the numerous interactions that have taken place between the participants of the “Unconventional Programming Paradigms” (UPP04) workshop⁴ [8] as well as the “The Grand Challenge in Non-Classical Computation International Workshop”⁵.

³Gerald L. Sussman, speaking about the programming of programmable materials [19].

⁴<http://upp.lami.univ-evry.fr>

⁵<http://www.cs.york.ac.uk/nature/workshop>

We would like to thank P. Dittrich at the University of Jena, P. Prusinkiewicz at the University of Calgary and Antoine Spicher at the University of Evry for stimulating discussions, thoughtful remarks and warm support. The comments of the anonymous reviewers have greatly improved the english of the paper.

REFERENCES

- [1] (2002) The P Systems Web Page. <http://psystems.disco.unimib.it/>.
- [2] (2004) The Organic Computing Page. <http://www.organic-computing.org>.
- [3] (1990) PPSN - Parallel Problem Solving from Nature. Proceedings published from 1994 as LNCS volumes. <http://ls11-www.cs.uni-dortmund.de/PPSN/>.
- [4] (1995) International Meeting on DNA Computing. Proceedings published from 1995 to 2000 as AMS DIMACS volume and then published as LNCS volume. <http://hagi.is.s.u-tokyo.ac.jp/dna/>.
- [5] (1995) IPCAT - Information Processing in Cells and Tissues. Proceedings published by World Scientific and as special issues of the Biosystems journal.
- [6] (1998) UMC - Unconventional Computation. Proceedings published at Springer. <http://www.cs.auckland.ac.nz/CDMTCS/conferences/uc/uc.html>.
- [7] Abelson, Allen, Coore, Hanson, Homsy, Knight, Nagpal, Rauch, Sussman and Weiss., (2000). Amorphous computing. *CACM: Communications of the ACM*, 43.
- [8] Banâtre, Jean-Pierre; Fradet, Pascal; Giavitto, Jean-Louis and Michel, Olivier; editors, (2005). *Unconventional Programming Paradigms*, Revised Selected and Invited Papers of the International Workshop UPP 2004, Le Mont-Saint-Michel, France. Springer-Verlag, LNCS, 3566.
- [9] Banâtre, Jean-Pierre and Le Métayer, Daniel., (1990). The GAMMA model and its discipline of programming. *Science of Computer Programming*, 15,(1): 55–77.
- [10] Benveniste, Albert; Caspi, Paul; Edwards, Stephen; Halbwachs, Nicolas; Le Guernic, Paul and de Simone, Robert., (2003). The synchronous languages twelve years later. *Proc. of the IEEE, Special issue on embedded systems*, 91,(1): 64–83.
- [11] Dijkstra, Edsger W., (1976). *A Discipline of Programming*. Prentice-Hall.
- [12] Dittrich, Petter; Ziegler, Jens and Banzhaf, Wolfgang., (2001). Artificial chemistries - a review. *Artificial Life*, 7,(3): 225–275.
- [13] Felleisen, Matthias., (1991). On the expressive power of programming languages. *Science of Computer Programming*, 17,(1–3): 35–75.
- [14] Feynman, Richard P., Hey, Anthony J. G. and Allen, Robin W., editors, (1996). *Feynman Lectures on Computation*. The Advanced Book Program. Addison-Wesley, Reading, MA.
- [15] Giavitto, Jean-Louis and Michel, Olivier., (2002). The topological structures of membrane computing. *Fundamenta Informaticae*, 49: 107–129.
- [16] Heiss, Janice., (2003). Coding from Scratch: A Conversation with Virtual Reality Pioneer Jaron Lanier. Sun Developer Network (SDN). Cf. http://java.sun.com/features/2003/01/lanier_qa1.html.
- [17] Horn, Paul., (2001). Autonomic computing: IBM's perspective on the state of information technology. Technical report, IBM Research. http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf.
- [18] Landin, Peter J., (1966). The next 700 programming languages. *Communications of the ACM*, 9,(3): 157–164. Originally presented at the Proceedings of the ACM Programming Language and Pragmatics Conference, 1965.

- [19] MIT Project Mac., (2003). The Amorphous Computing Home page.
<http://www.swiss.ai.mit.edu/projects/amorphous>.
- [20] Mitchell, John C., (1993). On abstraction and the expressive power of programming languages. In *TACS'91: Selected papers of the conference on Theoretical aspects of computer software*, 141–163, Amsterdam, The Netherlands, The Netherlands. Elsevier Science Publishers B. V.
- [21] Rieffel, Eleanor and Polak, Wolfgang., (2000). An introduction to quantum computing for non-physicists. *ACM Comput. Surv.*, 32,(3): 300–335.
- [22] Sussman, Gerald Jay., (1999). Robust design through diversity (position paper). In *Workshop on Amorphous Computing*, Cambridge. DARPA ITO – MIT Lab, Cf. <http://swiss.csail.mit.edu/projects/amorphous/workshop-sept-99/>.
- [23] Wirth, Niklaus., (1976). *Algorithms + Data Structures = Programs*. Prentice-Hall.