Unconventional Programming Paradigms

European Commission -

US National Science Foundation

Strategic Research Workshop

Unconventional Programming Paradigms

Mont Saint-Michel, France, 15-17 September 2004

Workshop Report

Pre-Proceedings

UPP'04

Unconventional Programming Paradigms

Challenges and Research Issues for New Programming Paradigms

15 - 17 September 2004
Le Mont Saint-Michel, France

Organized by

European Research Consortium for Informatics and Mathematics
ERCIM
www.ercim.org

Preface

Unconventional approaches of programming have long been developed in various niches and constitute a reservoir of alternative avenues to face the programming crisis. These new models of programming are also currently experiencing a renewed period of growth to face specific needs and new application domains. Examples are given by artificial chemistry, declarative flow programming, L-systems, P-systems, amorphous computing, visual programming systems, musical programming, multi-media interaction, etc. These approaches provide new abstractions and new notations or develop new ways of interacting with programs. They are implemented by embedding new and sophisticated data structures in a classical programming model (API), by extending an existing language with new constructs (to handle concurrency, exceptions, open environment, ...), by conceiving new software life cycles and program execution (aspect weaving, run-time compilation) or by relying on an entire new paradigm to specify a computation.

The practical applications of these new programming paradigms prompt researches into the expressivity, semantics and implementation of programming languages and systems architectures, as well as into the algorithmic complexity and optimization of programs.

The purpose of this workshop is to bring together researchers from the various communities working on wild and crazy ideas in programming languages to present their results, to foster fertilization between theory and practice, as well as to favor the dissemination and growth of new programming paradigms.

Apart from two invited talks, the contributions are dispatched into 5 tracks:

– Bio-inspired Computing
– Chemical Computing
– Amorphous Computing
– Autonomic Computing
– Generative Programming

The organizing committee
Summer 2004

Organization

UPP'04 is part of a series of strategic workshops to identify key research challenges and opportunities in Information Technology. For more information about this initiative, see http://www.ercim.org/EU-NSF/

Organizing Committee

Jean-Pierre Banâtre	Irisa/Université de Rennes I
Jean-Louis Giavitto	LaMI/Université d'Evry
Pascal Fradet	Inria Rhône-Alpes
Olivier Michel	LaMI/Université d'Evry

Track Leaders

George Păun	Bio-Inspired Computing
Peter Dittrich	Chemical Computing
Daniel Coore	Amorphous Computing
Manish Parashar	Autonomic Computing
Pierre Cointe	Generative Programming

Sponsoring Institutions

– Supported by the European Commissions Information Society Technologies Programme, Future and Emerging Technologies Activity, and the US National Science Foundation, Directorate for Computer and Information Science and Engineering.
– Organized by Ercim with additional support from Inria, Université d'Evry Val d'Essonne, Université de Rennes I, Microsoft Research, Lami, Irisa, Genopole, and Cnrs.

Table of Contents

Languages for Systems Biology

Luca Cardelli

Microsoft Research

I propose to study languages that can precisely and concisely represent biological processes such as the one described below. I give a very specific example, for concreteness and for shock value. But the range of phenomena and problems that fit in this endeavor is much larger. The domain is that of systems biology [13], which aims to represent not only cellular-level phenomena, such as the one below, but also phenomena at the level of tissues, organs, organisms, and colonies. Descriptive formalisms are needed to represent and relate many levels of abstraction.

The given example concerns the “algorithm” that a specific virus follows to reproduce. It is a sequence of steps that involve the dynamic merging and splitting of compartments, the transport of materials, and the transcription and interpretation of digital information. The algorithm is informally described in English below. What are appropriate languages and semantic models that can accurately and concisely describe such an algorithm, at a high level of abstraction but *in its entirety*? Formal modeling (e.g., at the level that can drive a simulator) is becoming of central importance in biology, where complex processes need to be analyzed for hypothesis testing. The area is increasing concerned with the discrete, although stochastic and perturbation-proof, processing of information.

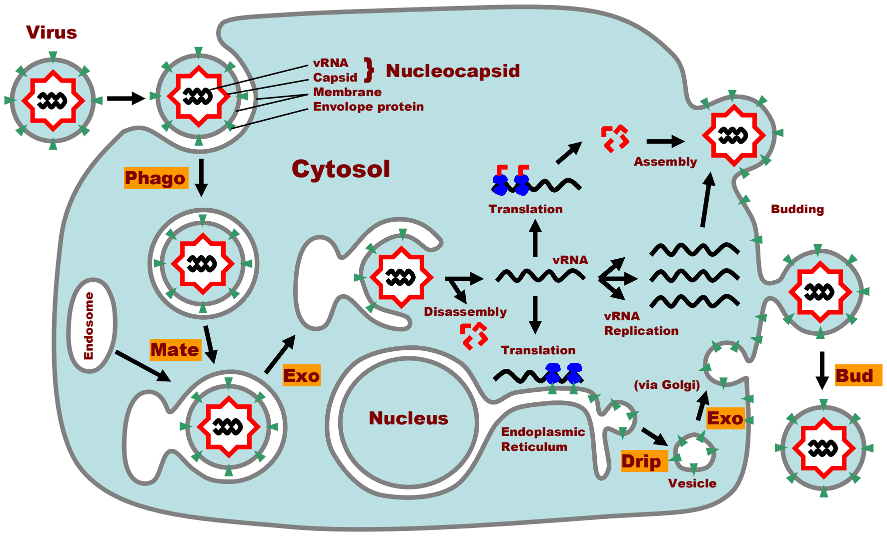


Figure 1 Semliki Forest Virus Infection and Reproduction ([1] p.279)

Figure 1. A virus is too big to cross a cellular membrane. It can either punch its RNA through the membrane or, as in this example, it can enter a cell by utilizing standard cellular endocytosis machinery. The virus consists of a capsid containing the viral RNA (the nucleocapsid). The

nucleocapsid is surrounded by a membrane that is similar to the cellular membrane (in fact, it is obtained from it "on the way out"). This membrane is however enriched with a special protein that plays a crucial trick on the cellular machinery, as we shall see shortly. The virus is brought into the cell by phagocytosis, wrapped in an additional membrane layer; this is part of a standard transport pathway into the cell. As part of that pathway, an endosome merges with the wrapped-up virus. At this point, usually, the endosome causes some reaction to happen in the material brought into the cell. In this case, though, the virus uses its special membrane protein to trigger an exocytosis step that deposits the naked nucleocapsid into the cytosol. The careful separation of internal and external substances that the cell usually maintains has now been subverted. The nucleocapsid is in direct contact with the inner workings of the cell, and can begin doing damage. First, the nucleocapsid disassembles itself, depositing the viral RNA into the cytosol. This vRNA then follows three distinct paths. First it is replicated (either by cellular proteins, or by proteins that come with the capsid), to provide the vRNA for more copies of the virus. The vRNA is also translated into proteins, again by standard cellular machinery. Some proteins are synthesized in the cytosol, and form the building blocks of the capsid: these self-assemble and incorporate a copy of the vRNA to form a nucleocapsid. The virus envelope protein is instead synthesized in the Endoplasmic Reticulum, and through various steps (through the Golgi apparatus) ends up lining transport vesicles that merge with the cellular membrane, along another standard transport pathway. Finally, the newly assembled nucleocapsid makes contact with sections of the cellular membrane that are now lined with the viral envelope protein, and buds out to recreate the initial virus structure outside the cell.

Are existing languages and semantic models adequate to represent these kinds of situations? Many classical approaches are relevant, but I believe the current answer must be: definitely not. Biologists are busy inventing their own abstact notations [8][9][10]. There are, of course, some proposals from computing as well [2][3][5][6][7]. The systems to be described are massively concurrent, heterogeneous, and asynchronous (notoriously the hardest ones to cope with in programming), with stochastic behavior and high resilience to drastic changes of environment conditions. What organizational principles make these systems work predictably? [11][12]

Answers to these questions should be of great interest to computing, for the organization of complex software systems. But that may come later: the proposal here is exclusively to model biological systems in order to understand how they work. The fundamental connection to computing (shared by systems biologists) is that many levels of organization are much more akin to software systems than to physical systems, both in hierarchical complexity and in algorithmic-like information-driven behavior. Hence the emphasis on the central role that languages may play.

References

[1] B.Alberts, D.Bray, J.Lewis, M.Raff, K.Roberts, J.D.Watson. **Molecular Biology of the Cell**. Third Edition, Garland.

[2] L.Cardelli. **Brane Calculi – Interactions of Biological Membranes**. http://research.microsoft.com/Users/luca/Papers/Brane%20Calculi.pdf.

[3] V.Danos and C.Laneve. **Formal Molecular Biology**. Theoretical Computer Science, to Appear.

[4] R.Milner. **Communicating and Mobile Systems: The π-Calculus**. Cambridge University Press, 1999.

[5] C.Priami. **The Stochastic pi-calculus**. The Computer Journal 38: 578-589, 1995.

[6] C.Priami, A.Regev, E.Shapiro, and W.Silverman. **Application of a stochastic name-passing calculus to representation and simulation of molecular processes**. Information Processing Letters, 80:25-31, 2001.

[7] A.Regev, E.M.Panina, W.Silverman, L.Cardelli, E.Shapiro. **BioAmbients: An Abstraction for Biological Compartments**. Theoretical Computer Science, to Appear.

[8] K. W. Kohn: **Molecular Interaction Map of the Mammalian Cell Cycle Control and DNA Repair Systems.** Molecular Biology of the Cell, 10(8):2703-34, Aug 1999.

[9] H. Kitano: **A graphical notation for biochemical networks**. BIOSILICO 1:169-176, 2003.

[10] **Systems Biology Markup Language.** http://www.sbml.org

[11] Hartwell LH, Hopfield JJ, Leibler S, Murray AW: **From molecular to modular cell biology.** Nature. 1999 Dec 2;402(6761 Suppl):C47-52.

[12] McAdams HH, Arkin A.: **It's a noisy business! Genetic regulation at the nanomolar scale**. Trends Genet. 1999 Feb;15(2):65-9.

[13] **4th International Conference on Systems Biology**. http://icsb2003.molecool.wustl.edu

Bio-Inspired Computing Paradigms (Natural Computing)

Gheorghe Păun

Institute of Mathematics of the Romanian Academy
PO Box 1-764, 7014700 Bucureşti, Romania, and
Research Group on Natural Computing
Department of Computer Science and Artificial Intelligence
University of Sevilla
Avda. Reina Mercedes s/n, 41012 Sevilla, Spain
E-mail: `george.paun@imar.ro, gpaun@us.es`

In some sense, the whole history of computer science is the history of a series of continuous attempts to discover, study, and, if possible, implement computing ideas, models, paradigms from the way nature – the humans included – computes. We do not enter here into the debate whether or not the processes taking place in nature are by themselves "computations", or we, *homo sapiens*, interpret them as computations, but we just recall the fact that when defining the computing model which is known now as *Turing machine* and which provides the standard by now definition of what is computable, A. Turing (in 1935 - 1936) explicitly wanted to abstract and model what a clerk in a bank is doing when computing with numbers. One decade later, McCullock, Pitts, Kleene founded the finite automata theory starting from modelling the neuron and the neural nets; still later, this led to the area called now *neural computing*. *Genetic algorithms* and evolutionary computing/programming are now well established (and much applied practically) areas of computer science. One decade ago, the history making Adleman's experiment of computing with DNA molecules was reported, proving that one can not only get inspired from biology for designing computers and algorithms for electronic computers, but one can also use a biological support (a bio-ware) for computing. In the last years, the search of computing ideas/models/paradigms in biology, in general in nature, became explicit and systematic under the general name of *natural computing*[1].

This important trend of computer science is not singular, many other areas of science and technology are scrutinizing biology in the hope – confirmed in many cases – that life has polished for billions of years numerous wonderful processes, tools, and machineries which can be imitated in domains completely separated from biology, such as materials and sensor technology, robotics, bionics, nanotechnology.

[1] As a proof of the popularity of this syntagm, it is of interest to point out that there are conferences with this topic explicitly included in their scope, a new journal with this name is published by Kluwer, a new series of the renown *Theoretical Computer Science* journal published by Elsevier is devoted to natural computing, a new series of books published by Springer-Verlag and a column in the *Bulletin of the European Association for Theoretical Computer Science* also have this name.

In order to see the (sometimes unexpected) benefits we can have in this framework, it is instructive to examine the case of genetic algorithms. Roughly speaking, they try to imitate the bio-evolution in solving optimization problems: the space of candidate solutions for a problem are encoded as "chromosomes" (strings of abstract symbols), which are evolved by means of cross-overing and point mutation operations, and selected from a generation to the next one by means of a fittness mapping; the trials to improve the fitness mapping continue until either no essential improvement is done for a number of steps, or until a given number of iterations are performed. The biological metaphors are numerous and obvious. What is not obvious (from a mathematical point of view) is why such a brute force approach – searching randomly the space of candidate solutions, with the search guided by random cross-overings and point mutations – is as successful as it happens to be (with a high probability, in many cases, the Genetic Algorithms provide a good enough solution in a large number of applications). The most convincing "explanation" is probably "because nature has used the same strategy in improving species". This kind of bio-mystical "explanation" provides a rather optimistic motivation for related researches.

A special mentioning deserves another "classic" area included nowadays in natural computing, namely neural computing. In short, the challenge is now to learn something useful from the brain organization, from the way the neurons are linked; the standard model consists of neuron-like computing agents (finite state machines, of very reduced capabilities), placed in the vertices of a net, with numerical weights on edges, aiming to compute a function; in a first phase, the net is "trained" for the task to carry out, and the weights are adjusted, then the net is used for solving a real problem. Pattern recognition problems are typical to be addressed via neural nets. The successes (and the promises) are comparable with those of genetic algorithms, without having a similarly wide range of applications. However, the brain remains such a misterious and efficient machinery that nobody can underestimate the progresses in any area trying to imitate the brain. (It also deserves to mention the rather interesting detail that Alan Turing himself, some years after introducing Turing machines, had a paper where he proposed a computing device in the form of a net of very simple computing units, able to learn, and then to solve an optimization problem – nothing else than neural computing *avant la lettre*. Unfortunately, his paper remained unpublished and was only recently reevaluated; see `http://www.AlanTuring.net` and [10] for details.)

Coming back to the history making Adleman's experiment mentioned above [1], it has the merit of opening (actually, confirming, because speculations about using DNA as a support for computations were made since several decades, while theoretical computing models inspired from the DNA structure and operations were already proposed in eighties, see, e.g., [6]) a completely new research vista: we can not only get inspired from biology for designing better algorithms for electronic computers, but we can also use a biological support (a bio-ware) for computing. Specifically, Adleman has solved in a lab, just handling DNA by techniques already standard in bio-chemistry, a computationally hard problem, the well-known Hamiltonian Path problem (whether or not in a given graph there is a path which visits all nodes, passing exactly once through each node). The problem is **NP**-complete, among those considered intractable for the usual computers,but Aldeman has solved it in linear time (the number of lab operations carried out was linear in terms of the number of nodes). The graph used in the experiment had only 7 nodes, a toy-problem by all means, while the actual working time was of seven days, but the *demo* (in terms of [5]) was convincing: we can compute using DNA!

It is important to note the fundamental novelty of this event: the objective was no longer to improve the use of standard electronic computers, as it was the goal of neural and evolutionary computing, but to have a principially new computer, based on using bio-molecules, in a bio-chemical manner. The great promise is to solve hard problems in a feasible time, by making use of the massive parallelism made possible by the very compact way of storing information on DNA molecules (bits at the molecular level, with some orders of efficiency over silicon supports). In this way, billions of "computing chips" can be accommodated in a tiny test tube, much more than on silicon. The possible (not yet very probable for the near future...) "DNA computer" also has other attractive features: energetical efficiency, reversibility, evolvability.

Another component of this general intellectual enterprise is membrane computing, which starts from the general observation that the cell is the smallest living thing, and at the same time it is a marvellous tiny machinery, with a complex structure, an intricate inner activity, and an exquisite relationship with its environment – the neighboring cells included. Then, the challenge is to find in the structure and the functioning of the cell those elements useful for computing. Distribution, parallelism, non-determinism, decentralization, (non)synchronization, coordination, communication, robustness, scalability, are only a few keywords related to this challenge. For instance, a problem which cannot be easily solved in terms of silicon engineering, but which was misteriously and very efficiently solved by nature at the level of the cell is related to the coordination of processes, the control pathways which keep the cell alive, without a high cost of coordination (in parallel computing the communication complexity is sometimes higher than the time and space complexity). Then, interesting questions appear in connection with the organization of cells into tissues, and this is also related to the way the neurons cooperate among them.

Similar issues are addressed by several other recent research directions belonging to natural computing, for instance, trying to learn computing ideas/models/paradigms from the way certain colonies of insects are organized and work together, the way bacteria populations develop in a given environment, the way flocks of birds maintain their "organization", the (amazing) way ciliates unscramble their chromosomes after reproduction, and so on. Most of these areas still wait for producing a *demo*, many of them are still in the stage of "craftsmanship", with *ad-hoc* ideas involved in *ad-hoc* models/tools handling *ad-hoc* problems, but the whole approach is both intellectually appealling and practically promising (sometimes through “by-products”, useful for biology, medicine, robotics, etc).

References

1. L.M. Adleman, Molecular Computation of Solutions to Combinatorial Problems, *Science*, 226 (November 1994), 1021–1024.
2. B. Alberts, A. Johnson, J. Lewis, M. Raff, K. Roberts, P. Walter, *Molecular Biology of the Cell*, 4th ed., Garland Science, New York, 2002.
3. J.A. Anderson, *An Introduction to Neural Networks*, The MIT Press, Cambridge, MA, 1996.
4. A. Ehrenfeucht, T. Harju, I. Petre, D.M. Prescott, G. Rozenberg, *Computations in Living Cells*, Springer-Verlag, Berlin, 2004.
5. J. Hartmanis, About the Nature of Computer Science, *Bulletin of the EATCS*, 53 (June 1994), 170–190.
6. T. Head, Formal Language Theory and DNA: An Analysis of the Generative Capacity of Specific Recombinant Behaviors, *Bulletin of Mathematical Biology*, 49 (1987), 737–759.
7. J.H. Koza, J.P. Rice, *Genetic Algorithms: The Movie*, MIT Press, Cambridge, Mass., 1992.
8. Gh. Păun, *Computing with Membranes: An Introduction*, Springer-Verlag, Berlin, 2002.
9. Gh. Păun, G. Rozenberg, A. Salomaa, *DNA Computing. New Computing Paradigms*, Springer-Verlag, Berlin, 1998.
10. C. Teuscher, *Alan Turing. Life and Legacy of a Great Thinker*, Springer-Verlag, Berlin, 2003.

Problem Solving by Evolution: One of Nature's Unconventional Programming Paradigms

Thomas Bäck[1][2], Ron Breukelaar[1], Lars Willmes[2]

[1] Universiteit Leiden, LIACS, P.O. Box 9512, 2300 RA Leiden, The Netherlands
{baeck,rbreukel}@liacs.nl
[2] NuTech Solutions GmbH, Martin Schmeißer Weg 15, 44227 Dortmund, Germany
{baeck,willmes}@nutechsolutions.de

Abstract. Evolving solutions rather than computing them certainly represents an unconventional programming approach. The general methodology of evolutionary computation has already been known in computer science since more than 40 years, but their utilization to program other algorithms is a more recent invention. In this paper, we outline the approach by giving an example where evolutionary algorithms serve to program cellular automata by designing rules for their evolution. The goal of the cellular automata designed by the evolutionary algorithm is a bitmap design problem, and the evolutionary algorithm indeed discovers rules for the CA which solve this problem efficiently.

1 Evolutionary Algorithms

Evolutionary Computation is the term for a subfield of Natural Computing that has emerged already in the 1960s from the idea to use principles of natural evolution as a paradigm for solving search and optimization problem in high-dimensional combinatorial or continuous search spaces. The algorithms within this field are commonly called evolutionary algorithms, the most widely known instances being genetic algorithms [6, 4, 5], genetic programming [7, 8], evolution strategies [11, 12, 14, 13], and evolutionary programming [3, 2]. A detailed introduction to all these algorithms can be found e.g. in the Handbook of Evolutionary Computation [1].

Evolutionary Computation today is a very active field involving fundamental research as well as a variety of applications in areas ranging from data analysis and machine learning to business processes, logistics and scheduling, technical engineering, and others. Across all these fields, evolutionary algorithms have convinced practitioners by the results obtained on hard problems that they are very powerful algorithms for such applications. The general working principle of all instances of evolutionary algorithms today is based on a program loop that involves simplified implementations of the operators mutation, recombination, selection, and fitness evaluation on a set of candidate solutions (often called a population of individuals) for a given problem. In this general setting, mutation corresponds to a modification of a single candidate solution, typically with a preference for small variations over large variations. Recombination corresponds to an exchange of components between two or more candidate solutions. Selection drives the evolutionary process towards populations of increasing average fitness by preferring better candidate solutions to proliferate with higher probability to the next generation than worse candidate solutions. By fitness evaluation, the calculation of a measure of goodness associated with candidate solutions is meant, i.e., the fitness function corresponds to the objective function of the optimization problem at hand.

This short paper does not intend to give a complete introduction to evolutionary algorithms, as there are many good introductory books on the topic available and evolutionary algorithms are, meanwhile, quite well known in the scientific community. Rather, we would like to briefly outline the general idea to use evolutionary algorithms to solve highly complex problems of parameterizing other algorithms, where the evolutionary algorithm is being used to find optimal parameters for another algorithm to perform its given task at hand as good as possible. One could also view this as an inverse design problem, i.e., a problem where the target design (behavior of the algorithm to be parameterized) is known, but the way to achieve this is unknown. The example we are choosing in this paper is the design of a rule for a 2 dimensional cellular automaton (CA) such that the cellular automaton solves a task at hand in an optimal way. We are dealing with 2 dimensional CAs where the cells have just binary states, i.e., can have a value of one or zero. The behavior of such a CA is fully characterized by a rule which, for each possible pattern of bit values in the local neighborhood of a cell (von Neumann neighborhood: the cell plus its four vertical and horizontal direct nearest neighbors; Moore neighborhood: the cell plus its 8 nearest neighbors, also including the diagonal cells), defines the state of this cell in the next iteration of the CAs evolution process. In the next section, we will explain the concept of a CA in some more detail. Section 3 reports experimental results of our approach with a 5 by 5 CA where the goal is to find rules which evolve from a standardized initial state of the CA to a target bit pattern, such that the rule rediscovers (i.e., inversely designs) this bit pattern. Finally, we give some conclusions from this work.

2 Cellular Automata

According to [15] Cellular Automata (CA) are mathematical idealizations of physical systems in which space and time are discrete, and physical quantities take on a finite set of discrete values. The simplest CA is one dimensional and looks a bit like an array of ones and zeros of a width N. The first position of the array is linked to the last position. In other words, defining a row of positions $C = \{a_1, a_2, ..., a_N\}$ where C is a CA of width N, then every a_n with $1 \leq n \leq N$ is connected to its left and right neighbors.

The neighborhood s_n of a_n is defined as the local set of positions with a distance to a_n along the connected chain which is no more than a certain radius (r). This for instance means that $s_2 = \{a_{148}, a_{149}, a_1, a_2, a_3, a_4, a_5\}$ for $r = 3$ and

$N = 149$. Please note that for one dimensional CA the size of the neighborhood is always equal to $2r + 1$.

There are different kinds of methods to change the values of a CA. The values can be altered all at the same time (synchronous) or at different times (asynchronous). Only synchronous CA were considered for this research. In the synchronous approach at time step t each value in the CA is recalculated according to the values of the neighborhood using a certain transition rule $\Theta : \{0,1\}^{2r+1} \rightarrow \{0,1\}, s_i \rightarrow \Theta(a_i)$. This rule can be viewed as a one-on-one mapping that defines an output value for every possible set of input values, the input values being the 'state' of a neighborhood. The state of a_n at time t is written as a_n^t, the state of s_n at time t as s_n^t and the state of the whole CA C at time t as C^t so that C^0 is the initial state (IC) and $a_n^{t+1} = \Theta(s_n^t)$. This means that C^{t+1} can be found by calculating a_n^{t+1} using Θ for all $1 \leq n \leq N$. Given $C^t = \{a_1^t, ..., a_N^t\}$, C^{t+1} can be defined as $\{\Theta(a_1^t), ..., \Theta(a_N^t)\}$.

Because $a_n \in \{0,1\}$ the number of possible states of s_n equals 2^{2r+1}. Because all possible binary representations of m where $0 \leq m < 2^{2r+1}$ can be mapped to a unique state of the neighborhood, Θ can be written as a row of ones and zeros $R = \{b_1, b_2, ..., b_{2^{2r+1}}\}$ where b_m is the output value of the rule for the input state that maps to the binary representation of $m - 1$. A rule therefore has a length that equals 2^{2r+1} and so there are $2^{2^{2r+1}}$ possible rules for a binary one dimensional CA. This is a huge number of possible rules (if $r = 3$ this sums up to about $3,4 \cdot 10^{28}$) each with a different behavior.

The two dimensional CA used in this paper do not differ much from the one dimensional CA discussed so far. Instead of a row of positions, C now consist of a grid of positions. The values are still only binary (0 or 1) and there still is only one transition rule for all the cells. The number of cells is still finite and therefore CA discussed here have a width, a height and borders.

The big difference between one dimensional and two dimensional CA is the rule definition. The neighborhood of these rules is two dimensional, because there are not only neighbors left and right of a cell, but also up and down. That means that if $r = 1$, s_n would consist of 5 positions, being the four directly adjacent plus a_n. This neighborhood is often called “the von Neumann neighborhood” after its inventor. The other well known neighborhood expands the Neumann neighborhood with the four positions diagonally adjacent to a_n and is called “the Moore neighborhood” also after its inventor.

Rules can be defined in the same rows of bits (R) as defined in the one dimensional case. For a Neumann neighborhood a rule can be defined with $2^5 = 32$ bits and a rule for a Moore neighborhood needs $2^9 = 512$ bits. This makes the Moore rule more powerful, for it has a bigger search space. Yet, this also means that searching in that space might take more time and finding anything might be a lot more difficult.

This research was inspired by earlier work in which transition rules for one dimensional CA were evolved to solve the Majority Problem [9, 10]. The genetic algorithm used here is a fairly simple algorithm with a binary representations of the rules, mutation by bit inversion, proportional selection, and without crossover.

In the next section experimental results on the bitmap problem are reported.

3 Using Evolutionary Algorithms to Program Cellular Automata

In preliminary experiments we tried different sizes of CA, but decided to concentrate on small square bitmaps with a width and a height of 5 cells. To make the problem harder and to stay in line with earlier experiments the CA has unconnected borders. To make the problem even more challenging the von Neumann neighborhood was chosen instead of the Moore neighborhood and therefore the s_n consist of 5 cells ($r = 1$) and a rule can be described with $2^5 = 32$ bits. The search space therefore is $2^{32} = 4294967296$. A bitmap of 32 b/w pixels would have the same number of possibilities, therefore this experiment is very challenging to say the least.

After testing different initial states, the 'single seed' state was chosen and defined as the state in which all the positions in the CA are 0 except the position $(\lfloor \text{width}/2 \rfloor, \lfloor \text{height}/2 \rfloor)$ which is 1. The theory being that this 'seed' should evolve or grow into the desired state in the same way as a single seed in nature can grow into a flower.

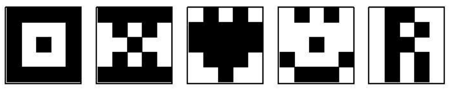


Fig. 1. The bitmaps used in the pattern generation experiment.

For this experiment only mutation was applied as an evolutionary operator. Mutation is performed by flipping every bit in the rule with a probability P_m. In this experiment $P_m = 1/\{\text{number of bits in a rule}\} = 1/32 = 0.03125$.

In trying to be as diverse as possible five totally different bitmaps were chosen, they are shown in figure 1. The algorithm was run 100 times for every bitmap for a maximum of 5000 generations. The algorithm was able to find a rule for all the bitmaps, but some bitmaps seemed a bit more difficult than others. Table 1 shows the number of successful rules for every bitmap. Note that symmetrical bitmaps seem to be easier to generate than asymmetric ones.

Although this experiment is fairly simple, it does show that a GA can be used to evolve transition rules in two dimensional CA that are able to generate patterns even with a simple von Neumann neighborhood. Figure 2 shows the behavior a few successful transition rules generated by the GA (in each row, the evolution of the CA is shown from left to right). Note that different transition rules can end up in the same desired state and have totally different iteration paths.

Ongoing experiments with larger CAs suggest that they do not differ much from these small ones, although the restrictions on what can be generated from a single-seed state using only a von Neumann neighborhood seem to be bigger when the size of the CA increases.

Table 1. Number of successful rules found per bitmap.

Bitmap	*Successful rules (out of a 100)*
"square"	80
"hourglass"	77
"heart"	35
"smiley"	7
"letter"	9

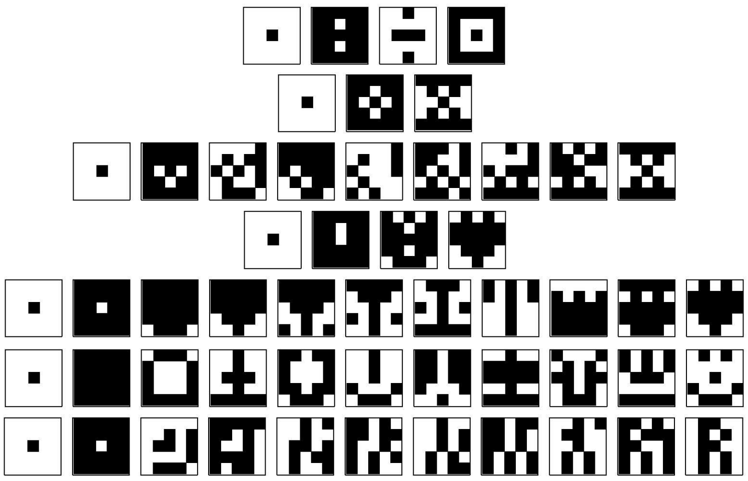


Fig. 2. This figure shows some iteration paths of successful transition rules.

4 Conclusions

The aim of the experiment reported in this paper was to demonstrate the capability of evolutionary algorithms, here a fairly standard genetic algorithm, to parameterize other methods such as, specifically, cellular automata. From the experimental results reported, one can conclude that this kind of inverse design of CAs is possible by means of evolutionary computation in a clear, straightforward, and very powerful way. The results clearly indicate that real world applications of CAs could also be tackled by this approach, and the unconventional programming of CAs by means of EAs is not only a possibility, but a useful and efficient method to parameterize this kind of algorithm.

References

1. Thomas Bäck, David B. Fogel, and Zbigniew Michalewicz, editors. *Handbook of Evolutionary Computation.* Institute of Physics Publishing and Oxford University Press, Bristol/New York, 1997.
2. David B. Fogel. *Evolutionary Computation – Toward a New Philosophy of Machine Intelligence.* IEEE Press, Piscataway, New York, 1995.
3. L. Fogel, Owens A., and M. Walsh. *Artificial Intelligence through Simulated Evolution.* John Wiley & Sons, 1966.
4. David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley, Reading, MA, 1989.
5. David E. Goldberg. *The Design of Innovation: Lessons from and for Competent Genetic Algorithms.* Kluwer Academic Publishers, 2002.
6. J.H. Holland. *Adaptation in Natural and Artificial Systems.* The University of Michigan Press, Ann Arbor, 1975.
7. John R. Koza. *Genetic Programming: On the Programming of Computers by Natural Selection.* MIT Press, Cambridge, MA, 1992.
8. John R. Koza, Martin A. Keane, Matthew J. Streeter, William Mydlowec, Jessen Yu, and Guido Lanza. *Genetic Programming IV: Routine Human-Competitive Machine Intelligence.* Kluwer Academic Publishers, 2003.
9. M. Mitchell and J.P. Crutchfield. The evolution of emergent computation. Technical report, Proceedings of the National Academy of Sciences, SFI Technical Report 94-03-012, 1994.
10. M. Mitchell, J.P. Crutchfield, and P.T. Hraber. Evolving cellular automata to perform computations: Mechanisms and impediments. *Physica D*, 75:361–391, 1994.
11. I. Rechenberg. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution.* Fromman-Holzboog Verlag, Stuttgart, 1973.
12. Ingo Rechenberg. *Evolutionsstrategie '94.* Frommann-Holzboog, Stuttgart, 1994.
13. Hans Paul Schwefel. *Evolution and Optimum Seeking.* Wiley, New York, 1995.
14. H.P. Schwefel. *Numerische Optimierung von Computer–Modellen mittels der Evolutionsstrategie*, volume 26 of *Interdisciplinary Systems Research.* Birkhäuser, Basel, 1977.
15. S. Wolfram. Statistical mechanics of cellular automata. *Reviews of Modern Physics*, 55, 1983.

Molecular Computations Using Self-Assembled DNA Nanostructures and Autonomous Motors

John H. Reif [*], Thomas H. LaBean, Sudheer Sahu, Hao Yan and Peng Yin

Abstract Self-assembly is the spontaneous self-ordering of substructures into superstructures driven by the selective affinity of the substructures. DNA provides a molecular scale material for programmable self-assembly, using the selective affinity of pairs of DNA strands to form DNA nanostructures. DNA self-assembly is the most advanced and versatile system that has been experimentally demonstrated for programmable construction of patterned systems on the molecular scale. The methodology of DNA self-assembly begins with the synthesis of single-strand DNA molecules that self-assemble into macromolecular building blocks called DNA tiles. These tiles have sticky ends that match the sticky ends of other DNA tiles, facilitating further assembly into larger structures known as DNA tiling lattices. In principle, DNA tiling assemblies can form any computable two or three-dimensional pattern, however complex, with the appropriate choice of the tiles' component DNA. Two-dimensional DNA tiling lattices composed of hundreds of thousands of tiles have been demonstrated experimentally. These assemblies can be used as scaffolding on which to position molecular electronics and robotics components with precision and specificity. This programmability renders the scaffolding have the patterning required for fabricating complex devices made of these components. We overview the evolution of DNA self-assembly techniques from pure theory, through simulation and design, and then to experimental practice. We will begin with an overview of theoretical models and algorithms for DNA lattice self-assembly. Then we describe our software for the simulation and design of DNA tiling assemblies and DNA nanomechanical devices. As an example, we discuss models and algorithms for the key problem of error control in DNA lattice self-assembly, as well as the computer simulation of these methods for error control. We will then briefly discuss our experimental laboratory demonstrations, including those using the designs derived by our software. These experimental demonstrations of DNA self-assembilies include the assembly of patterned objects at the molecular scale, the execution of molecular computations, and freely running autonomous DNA motors.

[*]Contact address: Department of Computer Science, Duke University, Box 90129, Durham, NC 27708-0129. E-mail: reif@cs.duke.edu.

1 Introduction

Self-assembly is the spontaneous self-ordering of substructures into superstructures driven by the selective affinity of the substructures. This paper focuses on a method for self-assembly known as *DNA self-assembly*, where DNA provides a molecular scale material for effecting this programmable self-assembly, using the selective affinity of pairs of DNA strands to form DNA nanostructures. Self-assembling nanostructures composed of DNA molecules offer great potential for bottom-up nanofabrication of materials and objects with smaller features than ever previously possible [13, 28, 33]. The methodology of DNA self-assembly begins with the synthesis of single-strand DNA molecules that self-assemble into macromolecular building blocks called DNA tiles. These tiles have sticky ends that match the sticky ends of other DNA tiles, facilitating further assembly into larger structures known as DNA tiling lattices. In principle, DNA tiling assemblies can be made to form any computable two- or three-dimensional pattern, however complex, with the appropriate choice of the tiles' component DNA.

DNA self-assembly is an emerging subfield of nanoscience with the development of its theoretical basis and a number of moderate to large-scale experimental demonstrations. Recent experimental results indicate that this technique is scalable. Periodic 2D DNA lattices have been successfully constructed with a variety of DNA tiles [14, 22, 43, 48]. These lattices are composed of up to hundreds of thousands of tiles. Molecular imaging devices such as atomic force microscopes and transmission electron microscopes allow visualization of these self-assembled two-dimensional DNA tiling lattices. These assemblies can be used as scaffolding on which to position molecular electronics and other components such as molecular sensors with precision and specificity. The programmability lets this scaffolding have the patterning required for fabricating complex devices made of these components. Potential applications of DNA self-assembly and scaffolding include nanoelectronics, biosensors, and programmable/autonomous molecular machines.

In addition to manufacturing DNA lattices, DNA has also been demonstrated to be a useful material for molecular computing systems [1, 3, 6, 19, 21] and mechanical devices [17, 20, 36, 50]. In particular, the self-assembly of DNA tiles can also be used as a powerful computational mechanism [15, 27, 39, 42], which in theory holds universal computing power [40]. See [25] for a more detailed survey of current experimental work in self-assembled DNA nanostructures. Also, see [26] and [28] for comprehensive surveys of the larger field of DNA computation (also known as biomolecular computation).

In this paper, we overview the evolution of DNA self-assembly techniques from pure theory, through simulation and design, and then to experimental practice. The rest of the paper is organized as follows. In Section 2, we overview the theoretical work in self-assembly. In Section 3, we describe software for the simulation and design of DNA nanostructures and motors. As a concrete example, in Section 4 we discuss error control, which we feel is a major theoretical and practical challenge remaining in the area of DNA self-assembly. Finally, in Section 5 we give a brief discussion of experimental practice in DNA nanostructures.

2 The Theory of Self-Assembly

This section overviews the emerging theory of self-assembly.

Domino Tiling Problems. The theoretical basis for self-assembly has its roots in *Domino Tiling Problems* (also known as Wang tilings) as defined by Wang [37]. For comprehensive text, see [9]. The input is a finite set of unit size square tiles. The sides of each square are labeled with symbols over a finite alphabet. Additional restrictions may include the initial placement of a subset of the these tiles, and the dimensions of the region where tiles must be placed. Assuming an arbitrarily large supply of each tile, the problem is to place the tiles, without rotation (a criterion that cannot apply to physical tiles), to completely fill the given region so that each pair of abutting tiles have identical symbols on their contacting sides.

Turing-universal and NP-Complete Self-assemblies. Domino tiling problems over an infinite domain with only a constant number of tiles were first proved by Berger to be undecidable [7]. This and subsequent proofs [7, 29] rely on constructions where tiling patterns simulate single-tape Turing machines or cellular arrays [40]. Winfree later showed that computation by self-assembly is Turing-universal [40] and so tiling self-assemblies can theoretically provide arbitrarily complex assemblies even with a constant number of distinct tile types. Winfree also demonstrated various families of assemblies which can be viewed as computing languages from families of the Chomsky hierarchy [39]. It has been proved that Domino tiling problems over polynomial-size regions are $\mathbf{NP}$-complete [16]. Subsequently, [39], [10, 11], and [15] proposed the use of self-assembly processes (in the context of DNA tiling and nanostructures) to solve $\mathbf{NP}$-complete combinatorial search problems such as SAT and graph coloring.

Program-size Complexity of Tiling Self-assemblies. The programming of tiling assemblies is determined simply by the set of tiles, their pads, and sometimes the choice of the initial seed tile (a special tile from which the growth of the assembly starts). A basic issue is the number of distinct tile types required to produce a specified tile assembly. The *program size complexity* of a specified tiling is the number of distinct tiles (with replacement) to produce it. Rothemund and Winfree showed that the assembly of an $n \times n$ size square can be done using $\Theta(\log n/ \log\log n)$ distinct tiles and that the largest square uniquely produced by a tiling of a given number of distinct tiles grows faster than any computable function [30]. Adleman recently gave program size complexity bounds for tree shaped assemblies [3].

Massively Parallel Computation by Tiling. Parallelism reveals itself in many ways in computation by self-assembly. Each superstructure may contain information representing a different calculation (*global parallelism*). Due to the extremely small size of DNA strands, as many as 10^{18} DNA tiling assemblies may be made simultaneously in a small test tube. Growth on each individual superstructure may also occur at many locations simultaneously via *local parallelism*. The *depth* of a tiling superstructure is the maximum number of self-assembly reactions experienced by any substructure (the depth of the graph of reaction events), and the *size* of a superstructure is the number of tiles it contains. Likewise we can define the number of layers for a superstructure. For example, a superstructure consisting of an array of $n \times m$ tiles, where $n > m$ has m layers. Tiling systems with low depth, small size, and few layers are considered desirable, motivating the search for efficient computations performed by such systems. Reif was the first to consider the parallel depth complexity of tiling assemblies and gave DNA self-assemblies of linear size and logarithmic depth for a number of fundamental problems (e.g., prefix computation, finite state automata simulation, and string fingerprinting, etc.) that form the basis for the design of many parallel algorithms [27]. Furthermore, [27] showed that these elementary operations can be combined to perform more complex computations, such as bitonic sorting and general circuit evaluation with polylog depth assemblies.

Linear Self-Assemblies. Tiling systems that produce only superstructures with k layers, for some constant k, are said to use *linear self-assembly*. [27] gave some simple linear tiling self-assemblies for integer addition as well as related operations (e.g., prefix XOR summing of n Boolean bits). Seeman's group demonstrated the first example of DNA computation using DNA tiling self-assembly [21], as described in Section 5. These linear tilings were refined in [44] to a class of String tilings that have been the basis for further DNA tiling experiments in [46] described in Section 5.

Kinetic Models of Tiling Self-assembly Processes. Domino tiling problems do not presume or require a specific process for tiling. Winfree first observed that self-assembly processes can be used for computation via the construction of DNA tiling lattices [38]. The sides of the tiles are assumed to have some methodology for selective affinity, which we call *pads*. Pads function as programmable binding domains, which hold together the tiles. Each pair of pads have specified binding strengths. The self-assembly process is initiated by a singleton tile (the *seed tile*) and proceeds by tiles binding together at their pads to form aggregates known as *tiling assemblies*. The preferential matching of tile pads facilitates the further assembly into tiling assemblies. Using the kinetic modeling techniques of physical chemistry, Winfree developed a kinetic model for the self-assembly of DNA tiles [41]. Following the classical literature of models for crystal assembly processes, Winfree considers assembly processes where the tiling assembly is only augmented by single tiles (known in crystallography as *monomers*) which bind to the assembly at their tile pads [38]. The likelihood of a particular tile binding at (or dissociating from) a particular site of the assembly is assumed to be a fixed probability dependent on that tile's concentration, the respective pad's binding affinity, and a temperature parameter. In addition, Adleman developed stochastic differential equation models for self-assembly of tiles and determined equilibrium probability distributions and convergence rates for some 1-dimensional self-assemblies [2, 4]. His model allowed for binding between subassemblies and assumed a fixed probability for tile binding events independent of the size of tile assemblies. Since the movement of tile assemblies may depend on their size (and thus mass), this model might in the future be refined to make the probability for tile binding events dependent on the size of tile assemblies.

Optimization of Tiling Assembly Processes. There are various techniques that may promote assembly processes in practice. One important technique is the tuning of the parameters (tile concentration, temperature, etc.) governing the kinetics of the process. Adleman considers the problem of determining tile concentrations for given assemblies and conjectures this problem is $\sharp\mathbf{P}$-complete [3]. Various other techniques may improve convergence rates to the intended assembly. A blockage of tiling assembly process can occur if an incorrect tile binds in an unintended location of the assembly. While such a tile may be dislodged by the kinetics of subsequent time steps, it still may slow down the convergence rate of the tiling assembly process to the intended final assembly. To reduce the possibility of blockages of tiling assembly processes, Reif proposed the use of distinct tile pads for distinct time steps during the assembly [27]. [27] also described the use of self-assembled tiling *nano-frames* to constrain the region of the tiling assemblies.

3 Simulation and Design Software

Software for Kinetic Simulation of Tiling Assembly Processes. Winfree developed software for discrete time simulation of the tiling assembly processes, using approximate probabilities for the insertion or removal of individual tiles from the assembly [41]. These simulations gave an approximation to the kinetics of self-assembly chemistry and provided some validation of the feasibility of tiling self-assembly processes. Using this software as a basis, our group developed an improved simulation software package (sped up by use of an improved method for computing on/off likelihood suggested by Winfree) with a Java interface for a number of example tilings, such as string tilings for integer addition and XOR computations. In spite of an extensive literature on the kinetics of the assembly of regular crystalline lattices, the fundamental thermodynamic and kinetic aspects of self-assembly of tiling assemblies are still not yet well understood. For example, the effect of distinct tile concentrations and different relative numbers of tiles is not yet known; probably it will require an application of Le Chatelier's principle.

Software for Kinetic Simulation of Nanomechanical Devices. We have developed a software to simulate autonomous nanomechanical DNA devices driven by ligase and restriction enzymes in a solution system. This software does discrete time simulation of the ligation and restriction events on the DNA duplex fragments of the nanomechanical device. The approximate probabilities of ligation is calculated based on the concentrations of individual DNA fragments present in the solution system. These simulations can provide insight to the kinetics of such nanomechanical systems. We have used this software to simulate a DNA walker and a universal DNA Turing machine.

Software for Design of DNA Lattices and Nanomechanical Devices. A major computational challenge in constructing DNA objects is to optimize the selection of DNA sequences so that the DNA strands can correctly assemble into desired DNA secondary structures. A commonly used software package, *Sequin*, was developed by Seeman, which uses the symmetry minimization algorithm. Sequin, though very useful, only provides a text-line interface and generally requires the user to step through the entire sequence selection process. Our lab recently developed a software package, *TileSoft*, which exploits an evolution algorithm and fully automates the sequence selection process. TileSoft also provides the user with a graphical user interface, on which DNA secondary structure and accompanying design constraints can be directly specified and the optimized sequence information can be pictorially displayed. TileSoft is initially designed to solve optimization problem for a set of multiple tiles, but can also be used to design individual DNA objects, such as DNA nanomechanical devices.

4 Error Control in DNA Tiling Assemblies

A chief challenge in DNA tiling self-assemblies is the control of assembly errors. This is particularly relavant to computational self-assemblies, which, with complex patterning at the molecular scale, are prone to a quite high rate of error, ranging from approximately between 0.5% to 5%, and the key barrier to large-scale experimental implementation of 2D computational DNA tilings exhibiting patterning is this significant error rate in the self-assembly process. The limitation and/or elimination of these errors in self-assembly is perhaps the single most important major challenge to nanostructure self-assembly.

There are a number of possible methods to decrease errors in DNA tilings:

(a) Annealing Temperature Optimization. This is a well known technique used in hybridization and also crystallization experiments. It can be used to decrease the defect rates at the expense of increased overall annealing time duration. In the context of DNA tiling lattices, the parameters for the temperature variation that minimize defects have not yet been determined.

(b) Error Control by Step-wise Assembly. Reif suggested the use of serial self-assembly to decrease errors in self-assembly [26].

(c) Error Control by Redundancy. There are a number of ways to introduce redundancy into a computational tiling assembly. In [24] we describe a simple method that can be developed for linear tiling assemblies: we replace each tile with a stack of three tiles executing the same function, and then add additional tiles that essentially ‘vote’ on the pad associations associated with these redundant tiles. This results in a tiling of increased complexity but still linear size. This error resistant design can easily be applied to the integer addition linear tiling described above, and similar redundancy methods may be applied to higher dimension tilings.

Work in 2003 by Winfree provided a method to decrease tiling self-assembly errors without decreasing the intrinsic error rate of assembling a single tile, however, his technique resulted in a final assembled structure that is four times the size of the original one [45].

Recently we have developed improved methods for compact error-resilient self-assembly of DNA tiling assemblies and analyzed them by probabilistic analysis, kinetic analysis and computer simulation; and plan to demonstrate these error-resilient self-assembly methods by a series of laboratory experiments. Our compact error-resilient tiling methods do not increase the size of the tiling assembly. They use 2-way overlay redundancy such that a single pad mismatch between a tile and its immediate neighbor forces at least one further pad mismatch between a pair of adjacent tiles in the neighborhood of this tile. Theoretical probabilistic analysis and empirical studies of the computer simulation of Sierpinsky Triangle tilings have been used to validate these error-resilient 2-way overlay redundancy tiling results; the analysis shows that the error rate is considerably reduced.

5 Experimental Progress

Self-assembled DNA Tiling Lattices. Seeman first pioneered DNA structure nanofabrication in the 1980s by assembling a multitude of DNA nanostructures (such as rings, cubes, and octahedrons) using DNA branched junctions [18, 31, 32]. However, these early DNA nanostructures were not very rigid. Later, rigid and stable DNA nanostructures (known as *tiles*) were developed. Typically they contain multiple DNA anti-parallel crossovers. Individual DNA tiles interact by annealing with other specific tiles via their ssDNA pads to self-assemble into desired superstructures known as *DNA tiling lattices*. These lattices can be either: *non-computational,* containing a fairly small number of distinct tile types in a periodic pattern; or *computational,* containing a larger number of tile types with more complicated association rules which perform computation during lattice assembly.

Periodic 2D DNA lattices have been successfully constructed with a variety of DNA tiles, for example, double-crossover (DX) DNA tiles [43], rhombus tiles [22], and triple-crossover (TX) tiles [14]. Our lab recently developed a "waffle"-like DNA lattice composed of a novel type of DNA tiles [48]. In addition, we have recently developed a new method for the assembly of aperiodic patterns [47]. This *directed nucleation assembly technique* uses a synthetic input DNA strand that encodes the required pattern, and specified tiles assemble around this input DNA strand, forming the required 1D or 2D lattice pattern.

The self-assembly of DNA tiles can also be used as a powerful computational mechanism [15, 27, 39, 42], and Winfree showed that two dimensional DNA tiling lattices can in theory be used to perform universal computation [39]. In collaboration with Seeman's lab, we have experimentally demonstrated a one-dimensional algorithmic self-assembly of triple-crossover DNA molecules (TX tiles), which performs a 4-step cumulative XOR computation [21]. In addition, we have recently demonstrated the first parallel computation with DNA tile self-assembly [46], exploiting the "string tile" model proposed by Winfree [44]. In our experimental implementation, the DX tiles, each encoding a whole entry of a bit wise XOR operation, associated with each other randomly in a parallel fashion, generating a molecular look-up table.

DNA Robotics. Existing DNA nanomechanical devices can exhibit motions such as open/close [34, 35, 50], extension/contraction [5, 8, 17], and rotation [20, 36]. These motions are mediated by external environmental changes such as the addition and removal of DNA fuel strands [5, 8, 17, 34, 35, 36, 50] or the change of ionic strength of the solution [20]. Our lab has recently constructed a robust sequence-dependent DNA nanomechanical actuator and have incorporated it into a 2D parallelogram DNA lattice [22]. The actuator can be switched reversibly between two states, mediated by the addition and removal of fuel DNA strands. In addition, we have achieved in our lab the construction of a unidirectional DNA walker that moves autonomously along a linear DNA track [49].

6 Conclusion

The self-assembly of DNA is a promising emerging method for molecular scale constructions and computations. We have overviewed the area of DNA tiling self-assemblies and noted a number of open problems. We have discussed the potential approaches for error-control in self-assembly techniques for DNA computation; particularly the use of error-resilient modified tiling methods. We have identified some technological impacts of DNA assemblies, such as using them as platform for constructing molecular electronic and robotic devices. Important future work includes further investigating potential broader technological impacts of DNA lattices. Many applications of DNA lattices rely on the development of appropriate attachment methods between DNA lattice and other nanoparticles, which itself is a key challenge in DNA based nanoscience.

Acknowledgement

This work was supported by DARPA/AFSOR Contract F30602-01-2-0561, NSF ITR Grant EIA-0086015, DARPA/NSF Grant CCR-9725021, NSF QuBIC Grant EIA-0218376, and NSF QuBIC Grant EIA-0218359

References

[1] L. Adleman. *Science*, 266:1021, 1994.
[2] L. Adleman. Technical Report 00-722, USC, 2000.

[3] L. Adleman *et al*. In *34th ACM STOC*, page 23. ACM Press, 2002.
[4] L. Adleman *et al*. Linear self-assemblies: equilibria, entropy, and convergence rates. manuscript.
[5] P. Alberti and J. Mergny. *PNAS*, 100:1569, 2003.
[6] Y. Benenson. *et al*. *Nature*, 414:430, 2001.
[7] R. Berger. *Memoirs of the American Mathematical Society*, 66, 1966.
[8] L. Feng *et al*. *Angew. Int. Ed.*, 42:4342, 2003.
[9] S. Grunbaum *et al*. *Tilings and Patterns* H Freeman and Company.
[10] N. Jonoska *et al*. In *Proceedings of IEEE ICEC'97*, page 261, 1997.
[11] N. Jonoska *et al*. *Computing with Bio-Molecules, theory and experiments*, page 93, 1998.
[12] T. H. LaBean *et al*. In *DNA Based Computers 5*, page 123, 2000.
[13] T. H. LaBean. *Introduction to Self-Assembling DNA Nanostructures for Computation and Nanofabrication*. World Scientific, 2002.
[14] T. H. LaBean *et al*. *JACS*, 122:1848, 2000.
[15] M. G. Lagoudakis *et al*. In *DNA Based Computers 5*, page 141, 2000.
[16] H. Lewis *et al*. *Elements of the Theory of Computation*. Prentice-Hall.
[17] J. Li *et al*. *Nanoletter*, 2:315, 2002.
[18] F. Liu, *et al*. *Biochemistry*, 38:2832, 1999.
[19] Q. Liu *et al*. *Nature*, 403:175, 2000.
[20] C. Mao *et al*. *Nature*, 397:144, 1999.
[21] C. Mao *et al*. *Nature*, 407:493, 2000.
[22] C. Mao *et al*. *JACS*, 121:5437, 1999.
[23] J. H. Reif. *Biomol. Computing, New Generation Computing*, 2002.
[24] J. H. Reif. In *29th ICALP, Mlaga, Spain*, page 1, 2002.
[25] J. H. Reif *et al*. *Lecture Notes in Computer Science*, 2054:173, 2001.
[26] J. H. Reif. In *First International Conference on Unconventional Models of Computation, Auckland, New Zealand*, page 72, 1998.
[27] J. H. Reif. In *DNA Based Computers 3*, 1997.
[28] J. H. Reif. *special issue on Bio-Computation, Computer and Scientific Engineering Magazine, IEEE Computer Society*, page 32, 2002.
[29] R. Robinson. *Inventiones Mathematicae*, 12:177, 1971.
[30] P. W. K. Rothemund *et al*. In *the 32nd ACM STOC*, page 459, 2000.
[31] N. Seeman *et al*. *J. Vac. Sci. Technol.*, 12:4:1895, 1994.
[32] N. C. Seeman. *Angew. Chem. Int. Edn Engl.*, 37:3220, 1998.
[33] N. C. Seeman. *Nature*, 421:427, 2003.
[34] F. Simmel *et al*. *Physical Review E*, 63:041913, 2001.
[35] F. Simmel *et al*. *Applied Physics Letters*, 80:883, 2002.
[36] H. Y. *et al*. *Nature*, 415:62, 2002.
[37] H. Wang. *Bell Systems Technical Journal*, 40:1, 1961.
[38] E. Winfree. In *DNA Based Computers 1*, page 187, 1995.
[39] E. Winfree. In *DNA Based Computers 1*, page 199, 1995.
[40] E. Winfree, *et al*. In *DNA Based Computers 2*, page 191, 1996.
[41] E. Winfree. PhD thesis, California Institute of Technology, 1998.
[42] E. Winfree. Technical Report 1988.22, Caltech, 1998.
[43] E. Winfree *et al*. *Nature*, 394:539, 1998.
[44] E. Winfree *et al*. In *DNA Based Computers 6*, page63, 2000.
[45] E. Winfree *et al*. In *DNA Based Computers 9*, 2003.
[46] H. Yan *et al*. *JACS*, 125:47, 2003.
[47] H. Yan *et al*. *PNAS*, 100:8103, 2003.
[48] H. Yan *et al*. *Science*, 301:1882, 2003.
[49] P. Yin *et al*. 2004. In preparation.
[50] B. Yurke *et al*. *Nature*, 406:605, 2000.

P systems: a modelling language

Marian Gheorghe

Department of Computer Science, Sheffield University
Regent Court, Portobello Street, Sheffield, S1 4DP, UK
`m.gheorghe@dcs.shef.ac.uk`

Motto: The first engineers were inspired by biology, as have been thousands of mythical (e.g., Daedalus) and real (e.g., Leonardo DaVinci) engineers since. John von Neumann studied cellular automaton and Alan Turing studied morphogenesys. More recently, several active research areas in computer science have emerged inspired by biology [11].

The interaction between biology and computer science is a two way process, both areas benefiting from this. We will refer first to computational biology which shows how biological models routed in computer science are defined and used to the benefit of biology.

Molecular biology has uncovered a multitude of biological data and properties (such as genome sequences), but these are not enough to understand and interpret biological systems. All biological systems consist of elements which interact. A two way approach leading towards understanding these systems is proposed: knowledge discovery based around data-mining, which extract significant patterns from huge amounts of experimental data, and simulation-based approach, which tests via experiments different hypothesis [14]. A crucial problem in this context is that research groups are able to exchange their models and create commonly accepted data repository (SBML [30] represents a standard open software for modelling and analysis).

Computational models used in biology are based on continuous mathematical approaches, but also on discrete methods. In this work only the second approach will be illustrated, for the former we just refer to [18]. The area of discrete modelling is quite very diverse and we only report here some of those approaches where the benefits are well-established.

Gene regulatory networks have been modelled using hybrid Petri net representations which seem to be closer to the biologists' intuition than other kinds of models used for the same purpose (electrical circuits, Boolean networks, differential equations) [17]. Metabolic pathways have been also considered with the same models [9] and tools dealing with hybrid Petri net specifications have been produced [19]. Petri net models have been introduced in this context because of their appropriate semantics, their intuitive graphical representation and their capabilities for mathematical analysis leading to studying various properties (boundness, liveness, S-invariants, T-invariants) [9]. As biological processes can be considered at many levels of detail, Petri net models allowing formal verification of formal models have been used as a complement to workflow models that can represent nesting and ordering of processes [24].

Statecharts have been employed as a general approach to modelling T cell maturation in the thymus, which is an illustrative example of the accumulation of experimental data into large disconnected datasets [7]. The main benefits being 'to direct new experimantation, to uncover gaps in the dataset, to provide new ways to think about and visualize data, to serve as a link between scientific papers and to highlight differences between hypotheses' [7].

Process algebra and stochastic π-calculus are used to model the behaviour of social insects [27] and biomolecular processes [25]. These models provide a useful formalism for understanding the relationship between individual behaviour of the components and the dynamic behaviour of the system. 'They combine computer simulation, Markov chain analysis and mean-field methods for analysis' [27].

On the other hand computer science benefited a lot in the past from biology in order to develop new concepts, algorithms, approaches, theories. Software engineering community can learn a great deal about building systems from the broader concepts surrounding biological cell programs and the strategies they use to robustly accomplish complex tasks such as development, healing and regeneration. A colony of cells cooperates to form a multicellular organism under the direction of a common genetic program. A swarm of bees cooperates to construct a hive. Engineers envision building self-organizing networks, creating vast distributed sensor systems, and even harnessing biological cells as a new computational substrate. These examples raise fundamental questions for the organization of computing systems: 'How do we engineer robust behaviour from the cooperation of vast numbers of unreliable parts, that are interconnected in unknown, irregular, and time-varying ways?' [28]. Emerging technologies, such as MEMs devices, are making it possible to bulk-manufacture millions of tiny computing agents with sensors and actuators, and embed these into materials and the environment. We would like to build novel applications from these technologies - programmable materials, smart dust, self-reconfiguring robots, self-assembling nanostructures -, to unearth the 'computational nature' of biological systems - cells, tissues, organs, social insects - and to infuse these applications and computational paradigms with the much needed robustness, scalability, ability to deal with uncertainty, noisy and dynamic environment that is present in natural systems.

A number of programming paradigms have emerged at the intersection of computer science with bio-chemical processes. Cell-based programming paradigm has been introduced in order to construct structures that heal themselves, showing a high capacity to withstand catastrophic failures of systems built that way [11]. Amorphous computing paradigm considers approaches for programming a medium of randomly distributed computing particles where the challenge is to produce programs that generate predictable behaviour from locally defined interactions [28]. The Growing Point Language (GPL) [6], Origami Shape Language (OSL) [20], and Paintable Programming [5] are examples of programming mechanisms for producing global self-organization using local cooperation. A language, called MGS, inspired by biological process transformations and used to simulate biological processes whose state space must be computed jointly with the global state of the system has been introduced as a unifying paradigm of several computational mechanisms [12].

Other more formal models of computation based on nature have been proposed. Gamma [3] and CHAM [4] represent two programming paradigms inspired by bio-chemical processes focusing on multiset transformations. New computational paradigms inspired by DNA processes have generated applicative [8] or more theoretical approaches [23], [13]. Very recently the role of compartments in biology has been considered by verious modelling approaches. Bioambients calculus [26], 'suitable for representing various aspects of molecular localization and compartmentalization' and membrane computing [21], an abstract model, in the framework of formal languages, of the cell and its functionality - similar with L systems [15] -, are just two examples of models dealing with compartments at the cellular level.

Membrane computing introduces the concept of P systems. Membranes are among the main elements of the living cells which separate the cell from its environment and split the content of the cell into small compartments by means of internal membranes. Each compartment contains its own enzymes and their specialized molecules. Therefore, a membrane structure has been identified as the main characteristic of every P system that is defined as a hierarchical arrangement of different membranes embedded in a unique main membrane that identify several distinct regions inside the system. Each region gets assigned a finite multiset of objects and a finite set of rules either modifying the objects or moving them from a place to another one. The structure of a P system is usually represented as a tree describing the hierarchical architecture based on membranes. A natural generalisation of the P system model can be obtained by considering P systems where the structure of the system is defined as an arbitrary graph. Each node in the graph represents a membrane, which gets assigned a multiset of objects and a set of rules for modifying these objects and communicating them alongside the edges of the graph [21]. These networks of communicating membranes are also known as tissue P systems because, from a biological point view, they can be interpreted as an abstract model of multicellular organisms. In such organisms, cells are specialized members of a multicellular community. They collaborate with each other to form a multitude of different tissues, arranged into organs performing various functions. This model may be also regarded as an abstraction of a population of bio-entities aggregated together in a more complex bio-unit. In this respect the model addresses not only the cellular and tissue levels, but also the case of various colonies of more complex organisms like ants, bees etc. Cells and populations of individuals are usually far from being stable; mechanisms enabling new cell components or individuals to be introduced, to update the links between them or to remove some elements play a fundamental role in the evolution of a biological system as an entity of interacting/cooperating components. The area of P systems as a branch of natural computing flourished in the last years especially on computational power aspects [22]. Very recently P systems have been used to model the behaviour of membrane proteins [2], metabolic algorithm and oscillatory phenomena [16]. Apart from a number of tools produced for various purposes [29], executable specifications of P systems using the sequential rewriting software tool Maude [1] have been also provided.

We have been working on modelling various biological phenomena and are going to present some results regarding the use of (population) P systems as a modelling language. One area where this model will be applied is that concerning the behaviour of Pharaoh's ant colonies where the interest was to model individual ants in such a way that the whole system will show a reliable self-organisation and exhibit emergent behaviour. P systems have also been used to specify the individual-based modelling approach to microbial ecology and evolution. This approach provides models for understanding adaptation and evolution among communities of self-replicating agents representing bacteria living in virtual computational ecosystems.

References

1. O. Andrei, G. Ciobanu, D. Lucanu (2004), Rewriting P systems in Maude, Fifth Workshop on Membrane Computing, Milano-Bicocca, Italy, 104-118.
2. I. Ardelean, D. Besozzi (2004), New proposals for the formalization of membrane proteins (at http://www.gcn.us.es).
3. J.P. Banatre, A. Coutant, D. Le Metayer (1987), Parallel machines for multiset transformation and their programming style, Technical report RR-0759, INRIA.
4. G. Berry, G. Boudol (1989), The chemical abstract machine, Technical report RR-1133, INRIA.
5. W. Butera (2002), Programming a paintable computer, MIT Media Lab, PhD thesis.
6. D. Coore (1998), Botanical computing: a developmental approach to generating interconnect topologies on an amorphous computer, MIT, PhD thesis.
7. S. Efroni, D. Harel, I. Cohen (2002), Towards rigorous comprehension of biological complexity: modeling, execution and visualization of thymic T cell maturation, Genome Res., 13(11):2485-2497.
8. C. Ferretti, G. Mauri, C. Zandron, eds. (2004)), DNA10, Tenth International Meeting on DNA Computing, Milano-Bicocca, Italy.
9. H. Genrich, R. Kuffner, K. Voss (2001), Executable Petri net models for the analysis of metabolic pathways, Int J STTT, 3:394-404.
10. S. George, D. Evans, L. Davidson (2002), A biologically inspired programming model for self-healing systems, WOSS'02, Charleston, USA.
11. S. George, D. Evans, S. Marchette (2003), A biological programming model for self-healing, First ACM Workshop on Survivable and Self-Regenerative Systems, Fairfax, USA.
12. J.-L. Giavitto, O. Michel (2001), MGS: a programming language for the transformations of topological collection, LaMI technical report, No 61-2001.
13. T. Harju, I. Petre, G. Rozenberg (2003), Gene assembly in ciliates: molecular operations, TUCS technical report, No 557.
14. H. Kitano (2002), Computational systems biology, Nature, 420:206- 210.

15. A. Lindenmayer (1968), Mathematical models of cellular interaction in development - Part I and II -, Journal of Theoretical Biology, 18:280-299, 300-315.
16. V. Manca (2004), On the dynamics of P systems, Fifth Workshop on Membrane Computing, Milano-Bicocca, Italy, 29-43.
17. H. Matsuno, A. Doi, M. Nagasaki, S. Miyano (2000), Hybrid Petri net representation of gene regulatory network, Proc. Pacific Symposium on Biocomputing '00, World-Scientific, 5:338-349.
18. J. Murray (2002), Mathematical Biology. I: An introduction, Springer, Berlin.
19. M Nagasaki, A. Doi, M. Sasaki, C. Savoie, H. Matsuno, S. Miyano (2002), Genomic Object Net in Java: A platform for biopathway modelling and simulation, Genome Informatics, 13:252-253.
20. R. Nagpal (2001), Programming self-assembly: constructing global shape using biologically-inspired local interactions and orgami mathematics, MIT, PhD thesis.
21. Gh. Paun (2002), Membrane computing. An introduction, Springer, Berlin.
22. Gh Paun (2004), From cells to computers: membrane computing - a quick overview, Proceedings of Tenth International Meeting on DNA Computing, Milano-Bicocca, Italy, 1-15.
23. Gh. Paun, G. Rozenberg, A. Salomaa (1998), DNA computing - new computing paradigms, Springer, Berlin.
24. M. Peleg, I. Yeh, R. Altman (2001), Modeling biological processes using workflow and Petri net models, Bioinformatics.
25. C. Priami, A. Regev, E. Shapiro, W. Silverman (2001), Application of a stochastic name-passing calculus to representation and simulation of molecular processes, Information Processing Letters, 88:25-31.
26. A. Regev, E Panina, W. Silverman, L. Cardelli, E. Shapiro (2003), BioAmbients: an abstraction for biological compartments (submitted).
27. D. Sumpter, G. Blanchard, D. Broomhead (2001), Ants and agents: a process algebra approach to modelling and colony behaviour, Bulletin of Mathematical Biology, 63:951-980.
28. http://www.swiss.ai.mit.edu/projects/amorphous/6.978/
29. http://psystems.disco.unimib.it/
30. http://www.sbml.org/

Chemical Blending with Particles, Cells, and Artificial Chemistries

Christof Teuscher

University of California, San Diego (UCSD), Department of Cognitive Science
9500 Gilman Drive, La Jolla, CA 92093-0515, USA
E-mail: christof@teuscher.ch, URL: www.teuscher.ch/christof

Abstract Whereas understanding the brain is arguably a major challenge of the 21^{st} century, making the next generation pervasive computing machines more lifelike probably equals to a similar challenge. Thereby, dealing with new physical computing substrates, new environments, and new applications is likely to be equally important as to provide new paradigms to organize, train, program and interact with such machines.
The goal of this contribution is to delineate a possible way to address the general scientific challenge of seeking for further progress and new metaphors in computer science by means of unconventional methods. I show that an amalgamation of (1) a particle-based and randomly interconnected substrate, (2) membrane systems, and (3) artificial chemistries in combination with (4) an unconventional organizational paradigm has interesting properties and allows to realize complex systems in an alternative way.

1 Motivation

Biologically-inspired computing (see for example [13, 22] for a general introduction), also commonly called *natural computing*, is an emerging and interdisciplinary area of research which is heavily relied on the fields of biology, computer science, and mathematics. It is the study of computational systems that use ideas and get inspiration from natural organisms to build large, complex, and dynamical systems. The principal goal of bio-inspired computing is to make machines more lifelike and to endow them with properties that traditional machines typically do not posses, such as for example adaptation, learning, evolution, growth, development, and fault-tolerance.

It is evident that biological organisms operate on completely different principles from those with which most engineers are familiar. Whereas life itself might be defined as a chemical system capable of self-reproduction and of evolving, computers constitute a fundamentally different environment where self-reproduction, evolution, and many other processes are all but naturally occurring. This makes it particularly difficult to “copy” nature, but bears many opportunities to get inspiration from it. A further difficulty often resides in the way how information in computers is represented and processed. Whereas the concepts of *computability* and *universal computation* are undoubtedly central to theoretical computer science, their importance might be questioned with regards to biologically-inspired computing machines and to biological organisms. The traditional concept of "computation" can in most cases not straightforwardly be applied to biological components or entire organisms and there is no evidence that such a system can compute "universally", on the contrary, biology seems to prefer highly specialized units. And whether the metaphor of the brain or mind as a digital computer is valuable is still much debated. Although we have experienced in the past that biological systems are generally difficult to describe and to model by algorithmic processes, there is little reason to believe that they "compute" beyond the algorithmic domain, since, at the bottom it is all just physical stuff doing what it must.

The quest for novel computing machines and concepts is motivated by the observation that fundamental progress in machine intelligence and artificial life seem to stagnate [3]. For example, one of the keys to machine intelligence is computers that learn in an open and unrestricted way, and we are still just scratching the surface of this problem. Connectionist models have been unable to faithfully model the nervous systems of even the simplest living things. Although the interconnectivity of the *C. elegans'* 302 neurons in the adult animal is known, it has—to the best of my knowledge—not been possible so far to mimic the worm's simple nervous system. One reason is that artificial neural neurons are gross oversimplifications of real neurons. Another fundamental problem is that parallel programming has failed to produce general methods for programming massively parallel systems. Our abilities to program complex systems are simply not keeping up with the desire to solve complex problems. And equally serious is the lack of systematic and formal approaches to gradually create hierarchical and complex systems that scale, although some attempts have been made (see Section also 4).

I believe that some of today's and upcoming computing challenges are best approached by unconventional paradigms instead of "straying" around the well-known concepts, changing the model's parameters, and putting hope in increasing computing power. Using Brooks words, I rather believe in "[. . .] something fundamental and currently unimagined in our models" [3] than in all other possibilities, although they might also play a role of course. We need new tools and new concepts that move seamlessly between brain, cognition, computation, growth, development, and self-organization—a finding that has also been made in a 2002 NSF/DOC-sponsored report [20]. This is definitely a transdisciplinary challenge with the need of simultaneous technical and conceptual innovations. Much debate has also been focused on what cognitive paradigms should be used in order to obtain a "intelligent" agents. This seems to somehow become an eternal and useless discussion since—as in many other domains—there is no best model. Different models are better for different things in different contexts [7].

The goal of my contribution is to delineate *one* possible way to address the general scientific challenge of seeking for further progress and new metaphors in computer science by means of unconventional methods. I show that an amalgamation of (1) a particle-based and randomly interconnected substrate, (2) membrane systems, and (3) artificial chemistries in combination with (4) an unconventional organizational paradigm has interesting properties and allows to realize complex systems in an alternative way. An overview on the goals and guiding principles shall be provided in the next section.

Please note that this is still ongoing work which extends and further develops some of the visions as first presented in [24]. The present paper is an extended abstract only.

2 Goals and General Guiding Principles

The visions and work presented in this contribution are mainly motivated by the following goals and guiding principles:

1. The system should be **robust** and **fault-tolerant**. Fault-tolerance is a major issue for tomorrow's computing machines and might potentially be beneficial for new manufacturing technologies, such as for example printing technologies to create cheap and flexible sheets of transistors (polymer electronics), for molecular electronics, and likely for nanotechnology. Further, it supports the general challenge of building "perfect machines out of imperfect components", which would help to drop manufacturing costs.
2. The system should allow for **hierarchies**. This allows to divide (top-down) and gradually create (bottom-up) complexity and to provide a mode of parallelism. It is also a means to optimize limited resources, to minimize the overhead of control, and to support fault-tolerance.
3. The system should potentially allow to continuously **adapt** and **invent** by means of an **unconventional method**. This helps to deal with cases unforeseen by the designer and with an ever changing environment. The reason for using an unconventional method is the hope to avoid drawbacks of existing methods and to obtain new and interesting properties.
4. Everything should be made as **local** and as **simple** as possible. This helps to obtain scalable, large and potentially cheaper systems, and prevents from using global controllers, which often represent a performance bottleneck and a source for failures.

These guiding principles directly lead to a number of methods and tools which amalgamation—as I will illustrate—will results in a computational system with interesting properties and application domains.

In order to obtain a fault-tolerant system, our architecture will rely on a irregular, inhomogeneous, asynchronously operating, and possibly imperfect particle-based substrate (see Section 3), not unlike an amorphous computer. Currently, the only possibility to build a perfect machine from imperfect components is to use redundancy (i.e., spare components), which clearly favors a particle-based implementation. I will also make extensive use of artificial chemistries, which represent—if appropriately used—an ideal means to compute in uncertain environments. Further, they have also been identified as potentially very promising for the perpetual creation of novelty [11], a feature that shall be later used to support adaptation and learning. In order to be able to build hierarchies, I will make use of cells and membranes throughout the system (see Section 4). Thereby, membrane systems will serve as a main source of inspiration. Finally, adaptation is achieved by a method inspired by *conceptual blending*, a framework of cognitive science that tries to explain how we deal with mental concepts. However, instead of dealing with mental concepts we will use artificial chemistries and membranes (see Section 5).

3 The Circuit Amorphous Computer Substrate

In a 1996 white paper first described the philosophy of *amorphous computing*[1] [1, 14]. Amorphous computing is the development of organizational principles and programming languages for obtaining coherent global behavior from the local cooperation of myriads of unreliable parts—all basically containing the same program—that are interconnected in unknown, irregular, and time-varying ways. In biology, this question has been recognized as fundamental in the context of animals (such as ants, bees, etc.) that cooperate and form organizations. Amorphous computing brings the question "down" to computing science and engineering. Using the metaphor of biology, the cells cooperate to form a multicellular organism (also called *programmable multitude*) under the direction of a genetic program shared by the members of the colony.

I will present a modified amorphous computer model which will serve as a substrate for the implementation of cellular membrane system hierarchies. The model is called *circuit amorphous computer* (CAC), since, compared to the original model, its interconnections are wire-based. The main component of each particle, called *amorphon*, is a "well-stirred" reactor (see Figure 1). I will also show that the distributed and randomly interconnected reactor network is particularly suited for building robust systems.

4 Cellular Computing Architectures and Hierarchies

The biological cell is a central building blocks of most living organisms. Reason why numerous more or less complex computational models and implementations have been proposed. Most cellular models are, however, so simple that the only thing they share with their biological counterpart is the name. In most cases, the reason for this simplicity is that "[t]he simplest living cell is so complex that supercomputer models may never simulate its behavior perfectly" [8].

Cellular automata (CA), originally conceived by Ulam and von Neumann in the 1940s to provide a formal framework for investigating the behavior of complex, extended systems [27], are probably the best known cellular machines. In the simplest case, their structure is completely uniform and regular. On the

[1] Website: `http://www.swiss.ai.mit.edu/projects/amorphous`

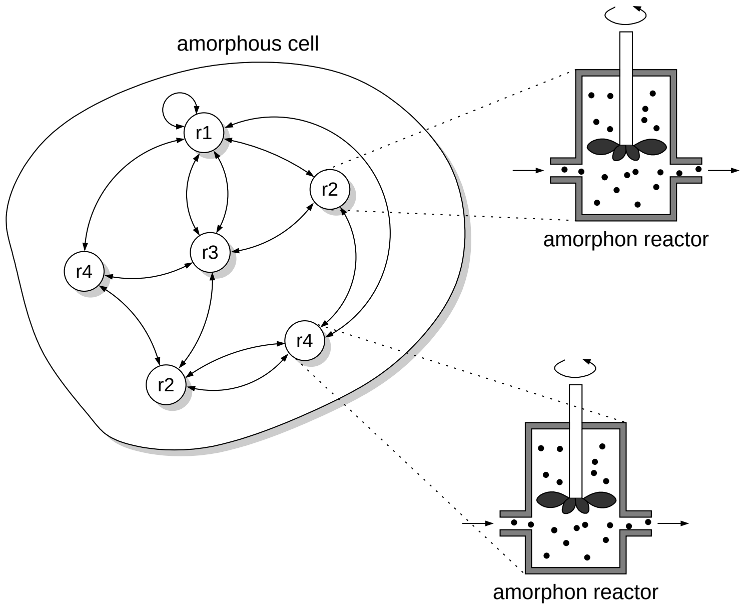


Figure 1. A cell is made up by a distributed network of "well-stirred" reactors.

other extreme, random boolean networks (RBNs)—as for example used by Kauffman [12]—have a random interconnection topology and non-uniform cells (i.e., each cell can have different rules). In both cases, there is no straightforward way to create hierarchies of cells. This is mainly due to the cell's simplicity and the lack of appropriate mechanisms.

In 1998, Paun initiated *P systems* (or *membrane computing*) [16, 17] as a highly parallel—though theoretical—computational model afar inspired by biochemistry and by some of the basic features of biological membranes. A typical P system (note that many variations exist) consists of cell-like membranes placed inside a unique "skin" membrane. Multisets of *objects*—usually strings of symbols—and a set of *evolution rules* are then placed inside the regions delimited by the membranes. The *artificial chemistry* [4] then evolves—or "computes"—according to the rules. A major drawback is, however, that all the rules have to be applied synchronously, which requires a global control signal. Additionally, membrane systems are usually engineered by hand since no methodology exists on how to set up a system for a given task.

P systems are particularly interesting since they allow to easily create hierarchies, which I consider a key issue for the creation of complex systems (see also Section 2). Hierarchical composition is ubiquitous in physical and biological systems: the nonlinear dynamics at each level of description generates emergent structure, and the nonlinear interactions among these structures provide a basis for the dynamics at the next higher level [21].

I will present how to create membrane systems on a circuit amorphous computer (Figure 2) and compare the approach to existing algorithms to create groups and hierarchies on amorphous computers.

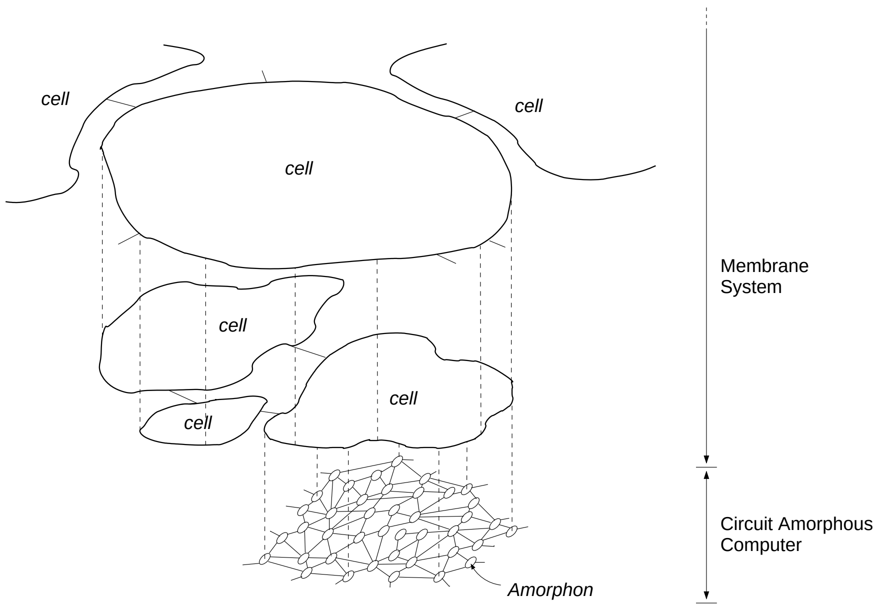


Figure 2. Implementing membrane systems on a circuit amorphous computer.

5 Chemical Blending

Conceptual blending (or *conceptual integration*) [5, 6] is a theory developed by Fauconnier and Turner about conceptual spaces and how they develop and proliferate as we talk and think. Conceptual spaces consist of elements and relations among them and are of course not directly instantiated in the brain, instead, they should be seen as a formalism invented by researchers to address and certain issues of their investigation. Conceptual spaces and blending are just a good tool to study meaning in natural language, metaphors, and concepts, but they are not suitable to talk about the structure of things [10]. Hence, Goguen and Harrell recently proposed a blending algorithm called *structural blending* [10], which also takes into account structure. A major drawback of blending is the difficulty to considered it as a real theory because of the lack of a formal approach. In recent years, however, several people worked on explicit computational frameworks [9, 10, 18, 19, 25, 26].

While all current approaches deal with concepts, I am interested in a very different approach. One of the fundamental problems of membrane systems and artificial chemistries in general is the fact that there does not exist any universal approach which allows to choose a set of objects (chemicals) and rules (reactions) able to solve a given task. Since this problem is basically equivalent to the challenge of programming a massively parallel machine, we should of course not expect astonishing breakthroughs.

The main motivations for investigating a blending-inspired approach applied to artificial chemistries are the following:

- How might “novelty” be created in a “smart” way in a membrane system?
- Can Fauconnier and Turner’s blending [6] be used in an elegant way to create “new” cells (in the sense of Paun’s membrane systems) from two already existing cells? The newly created cell should contain novel, but also already existing elements.
- How might blending be used as or inspire an organizing or optimizing principle for membrane systems?
- What are the benefits and drawbacks of such an approach?

I will illustrate these questions, provide examples (see Figure 3), and show how all this can be integrated. I will also delineate possible ways of how this approach could be used to obtain suitable artificial chemistries, particle-based systems, and reactive agents. The approach will also be compared to related work.

Acknowledgments

The author is supported by the Swiss National Science Foundation under grant PBEL2-104420.

References

1. H. Abelson, D. Allen, D. Coore, C. Hanson, E. Rauch, G. J. Sussman, and R. Weiss. Amorphous computing. *Communications of the ACM*, 43(5):74–82, May 2000.
2. E. Bilotta, D. Gross, T. Smith, T. Lenaerts, S. Bullock, H. H. Lund, J. Bird, R. Watson, P. Pantano, L. Pagliarini, H. Abbass, R. Standish, and M. A. Bedau, editors. *Alife VIII-Workshops. Workshop Proceedings of the* 8th *International Conference on the Simulation and Synthesis of Living Systems*. University of New South Wales, Australia, December 2002.
3. R. Brooks. The relationship between matter and life. *Nature*, 409:409–411, January 18 2001.
4. P. Dittrich, J. Ziegler, and W. Banzhaf. Artificial chemistries–a review. *Artificial Life*, 7(3):225–275, 2001.
5. G. Fauconnier and M. Turner. Conceptual integration networks. *Cognitive Science*, 22(2):133–187, April–June 1998.
6. G. Fauconnier and M. Turner. *The Way We Think: Conceptual Blending and the Mind’s Hidden Complexities*. Basic Books, 2002.
7. C. Gershenson. Cognitive paradigms: Which one is the best. *Cognitive Systems Research*, 5:135–156, 2004.

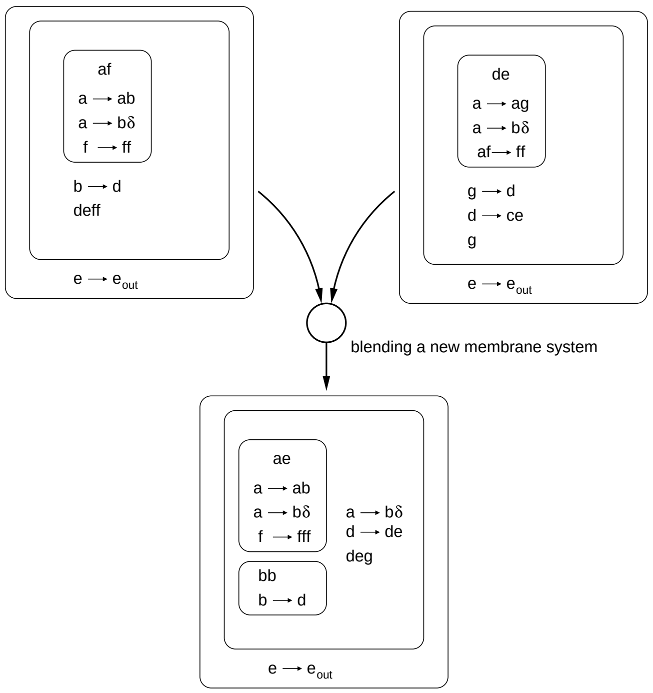


Figure 3. Illustration of the basic idea of chemical blending: the creation of a new cell which contains new structure but also elements from two existing cells. .

8. W. W. Gibbs. Cybernetic cells. *Scientific American*, 285(2):43–47, August 2001.
9. J. Goguen. An introduction to algebraic semiotics, with applications to user interface design. In Nehaniv [15], pages 242–291.
10. J. Goguen and F. Harrell. Foundations for active multimedia narrative: Semiotic spaces and structural blending. *Interaction Studies: Social Behaviour and Communication in Biological and Artificial Systems*, 2004. (To appear).
11. D. Gross and McMullin B. The creation of novelty in artificial chemistries. In Standish et al. [23], pages 400–408.
12. S. A. Kauffman. *The Origins of Order: Self–Organization and Selection in Evolution*. Oxford University Press, New York; Oxford, 1993.
13. D. Mange and M. Tomassini, editors. *Bio-Inspired Computing Machines: Towards Novel Computational Architectures*. Presses Polytechniques et Universitaires Romandes, Lausanne, Switzerland, 1998.
14. R. Nagpal. *Programmable Self-Assembly: Constructing Global Shape using Biologically-Inspired Local Interactions and Origami Mathematics*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, June 2001.
15. C. L. Nehaniv, editor. *Computation for Metaphor, Analogy and Agents*, volume 1562 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, Berlin, Heidelberg, 1999.

16. G. Paun. Computing with membranes. *Journal of Computer and System Sciences*, 61(1):108–143, 2000. First published in a TUCS Research Report, No 208, November 1998, `http://www.tucs.fi`.
17. G. Paun and G. Rozenberg. A guide to membrane computing. *Journal of Theoretical Computer Science*, 287(1):73–100, 2002.
18. F. C. Pereira and A. Cardoso. Knowledge integration with conceptual blending. In D. O'Donoghue, editor, *Proceedings of the Twelfth Irish Conference on Artificial Intelligence and Cognitive Science (AICS-2001)*, pages 33–42, Maynooth, Irland, September 2001. National University of Ireland, Department of Computer Science.
19. F. C. Pereira and A. Cardoso. The horse-bird creature generation experiment. *AISB Journal*, 2003.
20. M. C. Roco and W. S. Bainbridge, editors. *Converging Technologies for Improving Human Performance: Nanotechnology, Biotechnology, Information Technology and Cognitive Science*. World Technology Evaluation Center (WTEC), Arlington, Virginia, June 2002. NSF/DOC-sponsored report.
21. A. C. Scott. *Nonlinear Science: Emergence and Dynamics of Coherent Structures*. Oxford University Press, Oxford, 1999.
22. M. Sipper. *Machine Nature: The Coming Age of Bio-Inspired Computing*. McGraw-Hill, New York, 2002.
23. R. K. Standish, M. A. Bedau, and H. A. Abbass, editors. *Artificial Life VIII. Proceedings of the Eight International Conference on Artificial Life*. Complex Adaptive Systems Series. A Bradford Book, MIT Press, Cambridge, MA, 2003.
24. C. Teuscher. *Amorphous Membrane Blending: From Regular to Irregular Cellular Computing Machines*. PhD thesis, Swiss Federal Institute of Technology (EPFL), Lausanne, Switzerland, 2004. Thesis No 2925.
25. T. Veale and D. O'Donoghue. Computation and blending. *Cognitive Linguistics*, 11(3–4):253–281, 2000.
26. T. Veale, D. O'Donoghue, and M. T. Keane. Epistemological issues in metaphor comprehension: A comparison of three models and a new theory of metaphor. In *Proceedings of the International Cognitive Linguistics Conference (ICLA'95)*, University of New Mexico, Albuquerque, July 17–21 1995.
27. J. von Neumann. *Theory of Self-Reproducing Automata*. University of Illinois Press, Urbana, Illinois, 1966.

Membrane Systems: An Introduction

Gheorghe Păun

Institute of Mathematics of the Romanian Academy
PO Box 1-764, 7014700 Bucureşti, Romania, and
Research Group on Natural Computing
Department of Computer Science and Artificial Intelligence
University of Sevilla
Avda. Reina Mercedes s/n, 41012 Sevilla, Spain
E-mail: george.paun@imar.ro, gpaun@us.es

Abstract. Membrane Computing (MC) is part of the powerful trend in computer science known under the name of Natural Computing. Its goal is to abstract computing models from the structure and the functioning of the living cell. The present paper is a short and informal introduction to MC, presenting the basic ideas, the central (types of) results, and the main directions of research.

1 Membrane Computing – Starting From Cells

In the last decade, the continuous and mutually beneficial collaboration of informatics with biology became simply spectacular. Two landmark examples are the completion of the genome project, a great success of bio-informatics, of using computer science in biology, and the successful Adleman's experiment (1994) of using DNA molecules as a support for computing. The latter example is illustrative for the direction of research opposite to the traditional one, of using computers in biology: in Adleman's experiment, biological materials and techniques were used in order to solve a computational problem. This was the "official birth certificate" of what is now called DNA Computing, and this gave a decisive impulse to *Natural Computing*.

Membrane Computing is the youngest branch of Natural Computing. It starts from the observation that one of the most marvellous machineries evolved by nature are the cells. The cell is the smallest living unit, a microscopic "enterprise", with a complex structure, an intricate inner activity, and an exquisite relationship with its environment. Both substances, from ions to large macromolecules, and information are processed in a cell, according to involved reactions, organized in a robust and at the same time sensitive manner, having as the goal the processes themselves, the life itself of the cell and of the structures where the cells are included – organs, organisms, populations.

Thus, a double challenge emerged: to check whether or not the often made statements about the "computations" taking place in a cell (see, e.g., [2] and [3]) are mere metaphoras or they correspond to computations in the standard (mathematical) understanding of this term, and, more ambitiously, having in mind the encouraging experience of other branches of Natural Computing, to get inspired from the structure and the functioning of the living cell and define new computing models, possibly of interest for computer science, for computability in general.

Membrane computing emerged as an answer to this double challenge, proposing a series of models (actually, a general framework for devising models) inspired from the cell structure and functioning, as well as from the cell organization in tissue. These models, called P systems, were investigated as mathematical objects, with the main goals being of a (theoretical) computer science type: computation power (in comparison with Turing machines and their restrictions), and usefulness in solving computationally hard problems. The field (founded in 1998; the paper [4] was first circulated on web) simply flourished at this level. Comprehensive information can be found in the web page at `http://psystems.disco.unimib.it`; see also [5].

In this paper we discuss only the cell-like P systems, whose study is much more developed than that of tissue-like P systems or of neural-like P systems, only recently investigated in more details. In short, such a system consists of a hierarchical arrangement of *membranes* (understood as three-dimensional vesicles), which delimits *compartments* (also called *regions*), where abstract *objects* are placed. These objects correspond to the chemicals from the compartments of a cell, and they can be either unstructured, a case when they can be represented by symbols from a given alphabet, or structured. In the latter case, a possible representation of objects is by strings over a given alphabet. Here we discuss only the case of symbol-objects. Corresponding to the situation from reality, where the number of molecules from a given compartment matters, also in the case of objects from the regions of a P system we have to take into consideration their multiplicity, that is why we consider *multisets* of objects assigned to the regions of P systems. These objects evolve according to *rules*, which are also associated with the regions. The intuition is that these rules correspond to the chemical reactions from cell compartments and the reaction conditions are specific to each compartment, hence the evolution rules are localized. The rules say both how the objects are changed and how they can be moved (we say *communicated*) across membranes. By using these rules, we can change the *configuration* of a system (the multisets from its compartments); we say that we get a *transition* among system configurations. The way the rules are applied imitates again the biochemistry (but goes one further step towards computability): the reactions are done in *parallel*, and the objects to evolve and the rules by which they evolve are chosen in a *non-deterministic* manner, in such a way that the application of rules is maximal. A sequence of transitions forms a *computation*, and with computations which *halt* (reach a configuration where no rule is applicable) we associate a *result*, for instance, in the form of the multiset of objects present in the halting configuration in a specified membrane.

All these basic ingredients of a membrane computing system (a P system) will be discussed further below. This brief description is meant, on the one hand, to show the passage from the "real cell" to the "mathematical cell", as considered in membrane computing, and, on the other hand, to give a preliminary idea about the computing model we are investigating.

It is important to note at this stage the generality of the approach. We start from the cell, but the abstract model deals with very general notions: membranes interpreted as separators of regions, objects and rules assigned to regions; the basic data structure is the multiset; the rules are used in the non-deterministic maximally parallel manner, and in this way we get sequences of transitions, hence computations. In such terms, Membrane Computing can be interpreted as a *bio-inspired framework for distributed parallel processing of multisets.*

We close this introductory discussion by stressing the basic similarities and differences between MC and the other areas of Natural Computing. All these areas start from biological facts and abstract computing models. Neural and Evolutionary Computing are already implemented (rather successfuly, especially in the case of Evolutionary Computing) on the usual computer. DNA Computing has a bigger ambition, that of providing a new hardware, leading to bio-chips, to "wet computers". For MC it seems that the most realistic attempt for implementation is *in silico* (this started already to be a trend and some successes are already reported) rather than *in vitro* (no attempt was made yet).

2 The Basic Classes of P Systems

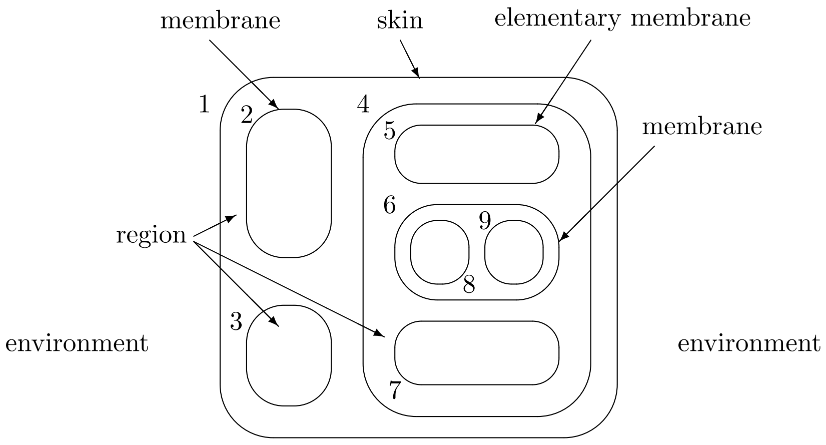
We introduce now the fundamental ideas of MC in a more precise way. What we look for is a computing device, and to this aim we need *data structures, operations* with these data structures, an *architecture* of our "computer", a systematic manner to define *computations* and *results* of computations.

Inspired from the cell structure and functioning, the basic elements of a *membrane system* (currently called *P system*) are (1) the *membrane structure* and the sets of (2) *evolution rules* which process (3) *multisets* of (4) *objects* placed in the compartments of the membrane structure.

A membrane structure is a hierarchically arranged set of membranes. A suggestive representation is as in the figure below. We distinguish the external membrane (corresponding to the plasma membrane and usually called the *skin* membrane) and several internal membranes (corresponding to the membranes present in a cell, around the nucleus, in Golgi apparatus, vesicles, etc); a membrane without any other membrane inside it is said to be *elementary.* Each membrane uniquely determines a compartment, also called *region*, the space delimited from above by it and from below by the membranes placed directly inside, if any exists.

In the basic class of P systems, each region contains a multiset of symbol-objects, which correspond to the chemicals swimming in a solution in a cell compartment; these chemicals are considered here as unstructured, that is why we describe them by symbols from a given alphabet.

The objects evolve by means of evolution rules, which are also localized, associated with the regions of the membrane structure. The rules correspond to the chemical reactions possible in the compartments of a cell. The typical form of such a rule is $aad \rightarrow (a, here)(b, out)(b, in)$, with the following meaning: two copies of object a and one copy of object d react and the reaction produces one copy of a and two copies of b; the new copy of a remains in the same region (indication *here*), one of the copies of b exits the compartment, going to the surrounding region (indication *out*) and the other enters one of the directly inner membranes (indication *in*). We say that the objects a, b, b are *communicated* as indicated by the commands associated with them in the right hand member of the rule. When an object exits a membrane, it will go to the surrounding compartment; in the case of the skin membrane this is the environment, hence the object is "lost", it never comes back into the system. If no inner membrane exists (that is, the rule is associated with an elementary membrane), then the indication *in* cannot be followed, and the rule cannot be applied.



The communication of objects through membranes reminds the fact that the biological membranes contain various (protein) channels through which the molecules can pass (in a passive way, due to concentration difference, or in an active way, with a consumption of energy), in a rather selective manner. The fact that the communication of objects from a compartment to a neighboring compartment is controlled by the "reaction rules" is attractive mathematically, but not quite realistic from a biological point of view, that is why there also were considered variants where the two processes are separated: the evolution is controlled by rules as above, without target indications, and the communication is controlled by specific rules (by symport/antiport rules – see below).

A rule as above, with several objects in its left hand member, is said to be *cooperative*; a particular case is that of *catalytic* rules, of the form $ca \rightarrow cx$, where a is an object and c is a catalyst, appearing only in such rules, never changing. A rule of the form $a \rightarrow x$, where a is an object, is called *non-cooperative.*

The rules associated with a compartment are applied to the objects from that compartment, in a *maximally parallel way*: all objects which can evolve by means of local rules should do it (we assign objects to rules, until no further assignment is possible). The used objects are "consumed", the newly produced objects are placed in the compartments of the membrane structure according to the communication commands assigned to them. The rules to be used and the objects to evolve are chosen in a non-deterministic manner. In turn, all compartments of the system evolve at the same time, synchronously (a common clock is assumed for all membranes). Thus, we have two layers of parallelism, one at the level of compartments and one at the level of the whole "cell".

A membrane structure and the multisets of objects from its compartments identify a *configuration* of a P system. By a non-deterministic maximally parallel use of rules as suggested above we pass to another configuration; such a step is called a *transition.* A sequence of transitions constitutes a *computation.* A computation is successful if it halts, it reaches a configuration where no rule can be applied to the existing objects. With a halting computation we can associate a *result* in various ways. The simplest possibility is to count the objects present in the halting configuration in a specified elementary membrane; this is called *internal output.* We can also count the objects which leave the system during the computation, and this is called *external output.* In both cases the result is a number. If we distinguish among different objects, then we can have as the result a vector of natural numbers. The objects which leave the system can also be arranged in a sequence according to the moments when they exit the skin membrane, and in this case the result is a string. This last possibility is worth emphasizing, because of the qualitative difference between the data structure used inside the system (multisets of objects, hence numbers) and the data structure of the result, which is a string, it contains a positional information, a syntax.

Because of the non-determinism of the application of rules, starting from an initial configuration, we can get several successful computations, hence several results. Thus, a P system *computes* (one also uses to say *generates*) a set of numbers, or a set of vectors of numbers, or a language.

Of course, the previous way of using the rules from the regions of a P system reminds the non-determinism and the (partial) parallelism from cell compartments, with the mentioning that the maximality of parallelism is mathematically oriented (rather useful in proofs); when using P systems as biological models, this feature should be replaced with more realistic features (e.g., reaction rates, probabilities, partial parallelism).

An important way to use a P system is the automata-like one: an *input* is introduced in a given region and this input is *accepted* if and only if the computation halts. This is the way for using P systems, for instance, in solving decidability problems.

We do not give here a formal definition of a P system. The reader interested in mathematical and bibliographical details can consult the mentioned monograph [5], as well as the relevant papers from the web bibliography mentioned above. Of course, when presenting a P system we have to specify: the alphabet of objects, the membrane structure (usually represented by a string of labelled matching parentheses), the multisets of objects present in each region of the system (represented by strings of symbol-objects, with the number of occurrences of a symbol in a string being the multiplicity of the object identified by that symbol in the multiset represented by the considered string), the sets of evolution rules associated with each region, as well as the indication about the way the output is defined.

Many modifications/extensions of the very basic model sketched above are discussed in the literature, but we do not mention them here. Instead, we only briefly discuss the interesting case of *computing by communication.*

In the systems described above, the symbol-objects were processed by multiset rewriting-like rules (some objects are transformed into other objects, which have associated communication targets). Coming closer to the trans-membrane transfer of molecules, we can consider purely communicative systems, based on the three classes of such transfer known in the biology of membranes: *uniport, symport*, and *antiport* (see [1] for details). Symport refers to the transport where two (or more) molecules pass together through a membrane in the same direction, antiport refers to the transport where two (or more) molecules pass through a membrane simultaneously, but in opposite directions, while the case when a molecule does not need a "partner" for a passage is referred to as uniport.

In terms of P systems, we can consider object processing rules of the following forms: a symport rule (associated with a membrane i) is of the form (ab, in) or (ab, out), stating that the objects a and b enter/exit together membrane i, while an antiport rule is of the form $(a, out; b, in)$, stating that, simultaneously, a exits and b enters membrane i.

A P system with symport/antiport rules has the same architecture as a system with multiset rewriting rules: alphabet of objects, membrane structure, initial multisets in the regions of the membrane structure, sets of rules associated with the membranes, possibly an output membrane – with one additional component, the set of objects present in the environment. This is an important detail: because by communication we do not create new objects, we need a supply of objects, in the environment, otherwise we are only able to handle a finite population of objects, those provided in the initial multiset. Also the functioning of a P system with symport/antiport rules is the same as for systems with multiset rewriting rules: the transition from a configuration to another configuration is done by applying the rules in a non-deterministic maximally parallel manner, to the objects available in the regions of the system and in the environment, as requested by the used rules. When a halting configuration is reached, we get a result, in a specified output membrane.

3 Computational Completeness; Universality

As we have mentioned before, many classes of P systems, combining various ingredients described above, are able of simulating Turing machines, hence they are *computationally complete*. Always, the proofs of results of this type are constructive, and this have an important consequence from the computability point of view: there are *universal* (hence *programmable*) P systems. In short, starting from a universal Turing machine (or an equivalent universal device), we get an equivalent universal P system. Among others, this implies that in the case of Turing complete classes of P systems, the hierarchy on the number of membranes always collapses (at most at the level of the universal P systems). Actually, the number of membranes sufficient in order to characterize the power of Turing machines by means of P systems is always rather small.

We only mention here two of the most interesting universality results:

1. P systems with symbol-objects with catalytic rules, using only two catalysts and two membranes, are universal.
2. P systems with symport/antiport rules of a rather restricted size (example: four membranes, symport rules of weight 2, and no antiport rules) are universal.

We can conclude that the compartmental computation in a cell-like membrane structure (using various ways of communicating among compartments) is rather powerful. The "computing cell" is a powerful "computer".

4 Computational Efficiency

The computational power is only one of the important questions to be dealt with when defining a new computing model. The other fundamental question concerns the computing *efficiency*. Because P systems are parallel computing devices, it is expected that they can solve hard problems in an efficient manner – and this expectation is confirmed for systems provided with ways for producing an exponential workspace in a linear way. Three main such possibilities have been considered so far in the literature, and *all of them were proven to lead to polynomial solutions to* **NP***-complete problems*: *membrane division*, *membrane creation*, and *string replication*. Using them, polynomial solutions to `SAT`, the Hamiltonian Path problem, the Node Covering problem, the problem of inverting one-way functions, the Subset-sum, and the Knapsack problems were reported (note that the last two are numerical problems, where the answer is not of the yes/no type, as in decidability problems) etc. Details can be found in ([5], [6], as well as in the web page of the domain.

Roughly speaking, the framework for dealing with complexity matters is that of *accepting P systems with input*: a family of P systems of a given type is constructed starting from a given problem, and an instance of the problem is introduced as an input in such systems; working in a deterministic mode (or a *confluent* mode: some non-determinism is allowed, provided that the branching converges after a while to a unique configuration), in a given time one of the answers yes/no is obtained, in the form of specific objects sent to the environment. The family of systems should be constructed in a uniform mode (starting from the size of instances) by a Turing machine, working a polynomial time.

This direction of research is very active at the present moment. More and more problems are considered, the membrane computing complexity classes are refined, characterizations of the **P**$\neq$**NP** conjecture were obtained in this framework, improvements are looked for. An important recent result concerns the fact that **PSPACE** was shown to be included in $\mathbf{PMC}_D$, the family of problems which can be solved in polynomial time by P systems with the possibility of dividing both elementary and non-elementary membranes [7].

5 Concluding Remarks

This paper was intended as a quick and general introduction to Membrane Computing, an invitation to this recent branch of Natural Computing.

The starting motivation of the area was to learn from the cell biology new ideas, models, paradigms useful for informatics – and we have informally presented a series of details of this type. The mathematical development was quite rapid, mainly with two types of results as the purpose: computational universality and computational efficiency. Recently, the domain started to be used as a framework for modelling processes from biology (but also from linguistics, management, computer graphics, etc), and this is rather important in view of the fact that P systems are (reductionistic, but flexible, easily scallable, algorithmic, intuitive) models of the whole cell; modelling the whole cell was often mentioned as an important challenge for the bio-computing in the near future – see, e.g., [8].

We have recalled only a few classes of P systems and only a few (types of) results. A detailed presentation of the domain is not only beyond the scope of this text, but also beyond the dimensions of a monograph; furthermore, the domain is fastly emerging, so that, the reader interested in any research direction, a more theoretical or a more practical one, is advised to follow the developments, for instance, through the web page mentioned in Section 2.

References

1. B. Alberts, A. Johnson, J. Lewis, M. Raff, K. Roberts, P. Walter, *Molecular Biology of the Cell*, 4th ed., Garland Science, New York, 2002.
2. D. Bray, Protein Molecules as Computational Elements in Living Cells. *Nature*, 376 (July 1995), 307–312.
3. S. Ji, The Cell as the Smallest DNA-based Molecular Computer, *BioSystems*, 52 (1999), 123–133.
4. Gh. Păun, Computing with Membranes, *Journal of Computer and System Sciences*, 61, 1 (2000), 108–143 (and Turku Center for Computer Science-TUCS Report 208, November 1998, `www.tucs.fi`).
5. Gh. Păun, *Computing with Membranes: An Introduction*, Springer, Berlin, 2002.
6. M. Pérez-Jiménez, A. Romero-Jiménez, F. Sancho-Caparrini, *Teoría de la Complejidad en Modelos de Computatión Celular con Membranas*, Kronos, Sevilla, 2002.
7. P. Sosik, The Computational Power of Cell Division in P Systems: Beating Down Parallel Computers? *Natural Computing*, 2, 3 (2003), 287–298.
8. M. Tomita, Whole-Cell Simulation: A Grand Challenge of the 21st Century, *Trends in Biotechnology*, 19 (2001), 205–210.

P Systems: Some Recent Results and Research Problems [⋆]

Oscar H. Ibarra

Department of Computer Science
University of California
Santa Barbara, CA 93106, USA

Abstract. We consider the following definition (different from the standard definition in the literature) of "maximal parallelism" in the application of evolution rules in a P system G: Let $R = \{r_1, ...r_k\}$ be the set of distinct rules in the system. G operates in maximal parallel mode if at each step of the computation, a maximal subset of R is applied, and at most one instance of any rule is used at every step (thus at most k rules are applicable at any step). We refer to this system as a maximally parallel system. We look at the computing power of P systems under three semantics of parallelism. For a positive integer $n \leq k$, define:

n-**Max-Parallel:** At each step, nondeterministically select a maximal subset of at most n rules in R to apply (this implies that no larger subset is applicable).
$\leq n$-**Parallel:** At each step, nondeterministically select any subset of at most n rules in R to apply.
n-**Parallel:** At each step, nondeterministically select any subset of exactly n rules in R to apply.

In all three cases, if any rule in the subset selected is not applicable, then the whole subset is not applicable. When $n = 1$, the three semantics reduce to the **Sequential** mode.

We look at two models of P systems that have been studied in the literature: catalytic systems and communicating P systems. We show that for these systems, n-**Max-Parallel** mode is strictly more powerful than any of the following three modes: **Sequential**, $\leq n$-**Parallel**, or n-**Parallel**. For example, it follows from a result in [6] that a 3-**Max Parallel** communicating P system is universal. However, under the three limited modes of parallelism, the system is equivalent to a vector addition system, which is known to only define a recursive set. This shows that "maximal parallelism" is key for the model to be universal.

We also briefly summarize other recent results concerning membrane hierarchy and computational complexity of P systems. Finally, we propose some problems for future research.

No proofs are given in this extended abstract. Some results presented here were obtained in collaboration with Zhe Dang and Hsu-Chun Yen.

Keywords: P system, catalytic Psystem, communicating P system, maximally parallel system, sequential system, limited parallelism, vector addition system, semilinear set, membrane hierarchy, computational complexity.

1 Introduction

There has been a flurry of research activities in the area of membrane computing (a branch of molecular computing) initiated five years ago by Gheorghe Paun [15]. Membrane computing identifies an unconventional computing model, namely a P system, from natural phenomena of cell evolutions and chemical reactions. Due to the built-in nature of maximal parallelism inherent in the model, P systems have a great potential for implementing massively concurrent systems in an efficient way that would allow us to solve currently intractable problems (in much the same way as the promise of quantum and DNA computing) once future bio-technology (or silicon-technology) gives way to a practical bio-realization (or chip-realization).

[⋆] This research was supported in part by NSF Grants IIS-0101134 and CCR02-08595.

The Institute for Scientific Information (ISI) has recently selected membrane computing as a fast "Emerging Research Front" in Computer Science (see http://esi-topics.com/erf/october2003.html). A P system is a computing model, which abstracts from the way the living cells process chemical compounds in their compartmental structure. Thus, regions defined by a membrane structure contain objects that evolve according to given rules. The objects can be described by symbols or by strings of symbols, in such a way that multisets of objects are placed in regions of the membrane structure. The membranes themselves are organized as a Venn diagram or a tree structure where one membrane may contain other membranes. By using the rules in a nondeterministic, maximally parallel manner, transitions between the system configurations can be obtained. A sequence of transitions shows how the system is evolving. Various ways of controlling the transfer of objects from a region to another and applying the rules, as well as possibilities to dissolve, divide or create membranes have been studied. P systems were introduced with the goal to abstract a new computing model from the structure and the functioning of the living cell (as a branch of the general effort of Natural Computing – to explore new models, ideas, paradigms from the way nature computes). Membrane computing has been quite successful: many models have been introduced, most of them Turing complete and/or able to solve computationally intractable problems (NP-complete, PSPACE-complete) in a feasible time (polynomial), by trading space for time. (See the P system website at http://psystems.disco.unimb/it for a large collection of papers in the area, and in particular the monograph [16].)

In the standard semantics of P systems [15, 16, 18], each evolution step of a system G is a result of applying all the rules in G in a maximally parallel manner. More precisely, starting from the initial configuration, w, the system goes through a sequence of configurations, where each configuration is derived from the directly preceding configuration in one step by the application of a multi-set of rules, which are chosen nondeterministically. For example, a catalytic rule $Ca \rightarrow Cv$ in membrane q is applicable if there is a catalyst C and an object (symbol) a in the preceding configuration in membrane q. The result of applying this rule is the evolution of v from a. If there is another occurrence of C and another occurrence of a, then the same rule or another rule with Ca on the left hand side can be applied. Thus, in general, the number of times a particular rule is applied at anyone step can be unbounded. We require that the application of the rules is maximal: all objects, from all membranes, which *can be* the subject of local evolution rules *have to* evolve simultaneously. Configuration z is reachable (from the starting configuration) if it appears in some execution sequence; z is halting if no rule is applicable on z.

In this paper, we study a different definition of maximal parallelism. Let G be a P system and $R = \{r_1, ..., r_k\}$ be the set of distinct rules in all the membranes. (Note that r_i uniquely specifies the membrane the rule belongs to.) We say that G operates in maximal parallel mode if at each step of the computation, a maximal subset of R is applied, and at most one instance of any rule is used at every step (thus at most k rules are applicable at any step). For example, if r_i is a catalytic rule $Ca \rightarrow Cv$ in membrane q and the current configuration has two C's and three a's in membrane q, then only one a can evolve into v. Of course, if there is another rule r_j, $Ca \rightarrow Cv'$, in membrane q, then the other a also evolves into v'. **Throughout the paper**, we will use this definition of maximal parallelism.

2 Catalytic System (CS)

We recall the definition of a multi-membrane catalytic system (CS) as defined in [15]. The membranes (regions) are organized in a hierarchical (tree) structure and are labeled 1, 2, .., m for some m, with the outermost membrane (the skin membrane) labeled 1. At the start of the computation, there is a distribution of *catalysts* and *noncatalysts* in the membranes (the distribution represents the initial configuration of the system). Each membrane may contain a finite set of catalytic rules of the form $Ca \rightarrow Cv$, where C is a catalyst, a is a noncatalyst, and v is a (possibly null) string of noncatalysts. When this rule is applied, the catalyst remains in the membrane the rule is in, symbol a is deleted from the membrane, and the symbols comprising v (if nonnull) are transported to other membranes in the following manner. Each symbol b in v has a designation or target, i.e., it is written b_x, where x can be $here$, out, or in_j. The designation $here$ means that the object b remains in the membrane containing it (we usually omit this target, when it is understood). The designation out means that the object is transported to the membrane directly enclosing the membrane that contains the object; however, we do not allow any object to be transported out of the skin membrane. The designation in_j means that the object is moved into a membrane, labeled j, that is directly enclosed by the membrane that contains the object.

It is important to note that our definition of catalytic system is different from what is usually called catalytic system in the literature. Here, we do not allow rules without catalysts, i.e., rules of the form $a \to v$. Thus our systems use only purely catalytic rules.

Suppose S is a CS with m membranes. Let $\{a_1, ..., a_n\}$ be the set of noncatalyst symbols (objects) that can occur in the configurations of S. Let $w = (w_1, ..., w_m)$ be the initial configuration, where w_i represents the catalysts and noncatalysts in membrane i. (Note that w_i can be null.) Each reachable configuration of S is an nm-tuple $(v_1, ..., v_m)$, where v_i is an n-tuple representing the multiplicities of the symbols $a_1, ..., a_n$ in membrane i. Note that we do not include the catalysts in considering the configuration as they are not changed (i.e., they remain in the membranes containing them, and their numbers remain the same during the computation). Hence the set of all reachable configurations of S, denoted by $R(S)$ is a subset of $\mathbf{N}^{mn}$. The set of all halting reachable configurations is denoted by $R_h(S)$.

It is known that for any set $Q \subseteq \mathbf{N}^n$ that can be accepted by a Turing machine, we can construct a 1-membrane CS G with only purely catalytic rules such that $R_h(G) = Q$ [19, 20, 5]. In fact, [5] shows that three distinct catalysts (where each catalyst appears exactly once in the initial configuration) are already sufficient for universality. Thus, in general, a 3-**Max-Parallel** 1-membrane CS can define a nonrecursive reachability set.

2.1 Sequential CS (Zero Parallelism)

In a sequential CS, each step of the computation consists of an application of a single nondeterministically chosen rule, i.e., the membrane and rule within the membrane to apply are chosen nondeterministically. Thus, the computation of the CS has no parallelism at all. It turns out that sequential CS's are much weaker. They define exactly the semilinear sets.

We need the definition of a vector addition system. An n-dimensional *vector addition system* (VAS) is a pair $G = \langle x, W \rangle$, where $x \in \mathbf{N}^n$ is called the *start point* (or *start vector*) and W is a finite set of vectors in $\mathbf{Z}^n$, where $\mathbf{Z}$ is the set of all integers (positive, negative, zero). The *reachability set* of the VAS $\langle x, W \rangle$ is the set $R(G) = \{z \mid \text{ for some } j, z = x + v_1 + ... + v_j, \text{ where for all } 1 \le i \le j, \text{ each } v_i \in W \text{ and } x + v_1 + ... + v_i \ge 0\}$. The *halting reachability set* $R_h(G) = \{z \mid z \in R(G), z + v \not\ge 0 \text{ for every } v \text{ in } W\}$.

A VAS $G = \langle x, W \rangle$, where each vector in W is in $\mathbf{N}^n$ (i.e., has nonnegative components) generates a *linear set*. Any finite union of linear sets is called a *semilinear set*.

An n-dimensional *vector addition system with states* (VASS) is a VAS $\langle x, W \rangle$ together with a finite set T of transitions of the form $p \to (q, v)$, where q and p are states and v is in W. The meaning is that such a transition can be applied at point y in state p and yields the point $y + v$ in state q, provided that $y + v \ge 0$. The VASS is specified by $G = \langle x, T, p_0 \rangle$, where p_0 is the starting state. The *reachability set* is $R(G) = \{z \mid \text{ for some } j, z = x + v_1 + ... + v_j, \text{ where for all } 1 \le i \le j, p_{i-1} \to (p_i, v_i) \in T, \text{ and } x + v_1 + ... + v_i \ge 0\}$. The *reachability problem* for a VASS (respectively, VAS) G is to determine, given a vector y, whether y is in $R(G)$. The *equivalence problem* is to determine given two VASS (respectively, VAS) G and G', whether $R(G) = R(G')$. Similarly, one can define the reachability problem and equivalence problem for halting configurations.

The following summarizes the known results concerning VAS and VASS [22, 7, 1, 8, 13]:

Theorem 1.
1. *Let G be an n-dimensional VASS. We can effectively construct an $(n + 3)$-dimensional VAS G' that simulates G.*
2. *If G is a 2-dimensional VASS G, then $R(G)$ is an effectively computable semilinear set.*
3. *There is a 3-dimensional VASS G such that $R(G)$ is not semilinear.*
4. *If G is a 5-dimensional VAS G, then $R(G)$ is an effectively computable semilinear set.*
5. *There is a 6-dimensional VAS G such that $R(G)$ is not semilinear.*
6. *The reachability problem for VASS (and hence also for VAS) is decidable.*

7. *The equivalence problem for VAS (and hence also for VASS) is undecidable.*

Clearly, it follows from part 6 of the theorem above that the halting reachability problem for VASS (respectively, VAS) is decidable.

A *communication-free VAS* is a VAS where in every transition, at most one component is negative, and if negative, its value is -1. They are equivalent to communication-free Petri nets, which are also equivalent to commutative context-free grammars [3, 9]. It is known that they have effectively computable semilinear reachability sets [3].

Our first result shows that a sequential CS is weaker than a maximally parallel CS.

Theorem 2. *The following are equivalent: communication-free VAS, sequential multi-membrane CS, sequential 1-membrane CS.*

Corollary 1.
1. *If S is a sequential multi-membrane CS, then $R(S)$ and $R_h(S)$ are effectively computable semilinear sets.*
2. *The reachability problem (whether a given configuration is reachable) for sequential multi-membrane CS is NP-complete.*

2.2 CS Under Limited Parallelism

Here we look at the computing power of the CS under three semantics of parallelism. Let $R = \{R_1, ..., R_k\}$ be the set of rules of the CS. For a positive integer $n \leq k$, define:

1. n**-Max-Parallel:** At each step, nonderministically select a maximal subset of at most n rules in R to apply (this implies that no larger subset is applicable).

2. $\leq n$**-Parallel:** At each step, nondeterministically select any subset of at most n rules in R to apply.

3. n**-Parallel:** At each step, nondeterministically select any subset of exactly n rules in R to apply.

In all three cases above, if any rule in the set selected is not applicable, then the whole set is not applicable. Note that when $n = 1$, the three semantics reduce to the **Sequential** mode.

Theorem 3. *For $n = 3$, a 1-membrane CS operating under the n-**Max-Parallel** mode can define a recursively enumerable set. For any n, a multi-membrane CS operating under $\leq n$-**Parallel** mode or n-**Parallel** mode can be simulated by a VASS (= VAS).*

2.3 Simple Cooperative 1-Membrane System

Now consider the case when the 1-membrane CS has only one catalyst C with initial configuration $C^k x$ for some k and string x of noncatalysts. Thus, there are k copies of the same catalyst in the initial configuration. The rules allowed are of the form $Ca \to v$ or of the form $Caa \to Cv$, i.e., C catalyzes two copies of an object. This system is equivalent to a special form of cooperative P system [15, 16]. A simple cooperative system (SCS) is a P system where the rules allowed are of the form $a \to v$ or of the form $aa \to v$. Moreover, there is some fixed integer k such that the system operates in maximally parallel mode, but uses no more that k rules in any step. Clearly, the two systems are equivalent.

Theorem 4. *A 1-membrane SCS operating in k-maximally parallel mode can simulate a Turing machine when k is at least* 9.

3 Communicating P Sytem (CPS)

A communication P System (CPS) has rules of the form form (see [19]):

1. $a \to a_x$
2. $ab \to a_x b_y$
3. $ab \to a_x b_y c_{come}$

where x, y can be $here$, out, or in_j. As before, $here$ means that the object remains in the membrane containing it, out means that the object is transported to the membrane directly enclosing the membrane that contains the object (or to the environment if the object is in the skin membrane), and $come$ can only occur within the outermost region (i.e., skin membrane), and it means import the object from the environment. The designation in_j means that the object is moved into a membrane, labeled j, that is directly enclosed by the membrane that contains the object.

3.1 Sequential 1-Membrane CPS

First we consider the case when there is only one membrane (the skin membrane). The computation is *sequential* in that at each step there is only one application of a rule (to one instance). So, e.g., if nondeterministically a rule like $ab \to a_{here} b_{out} c_{come}$ is chosen, then there must be at least one a and one b in the membrane. After the step, a remains in the membrane, b is thrown out of the membrane, and c comes into the membrane. There may be several a's and b's, but only one application of the rule is applied. Thus, there is *no* parallelism involved. The computation halts when there is no applicable rule. Again, we are only interested in the multiplicities of the objects when the system halts.

We shall see below that a 1-membrane CPS can be simulated by a VASS (= VAS). However, the converse is not true:

Theorem 5. *The set of (halting) reachable configurations of a sequential 1-membrane CPS is a semilinear set.*

3.2 Sequential 1-Membrane Extended CPS (ECPS)

Interestingly, if we generalize the rules of a 1-membrane CPS slightly the extended system becomes equivalent to a VASS. Define an extended CPS (ECPS) by allowing rules of the form:

1. $a \to a_x$
2. $ab \to a_x b_y$
3. $ab \to a_x b_y c_{come}$
4. $ab \to a_x b_y c_{come} d_{come}$

(i.e., by adding rules of type 4).

Theorem 6. *Sequential 1-membrane ECPS and VASS are equivalent.*

We can generalize rules of an ECPS further as follows:

1. $a_{i_1} ... a_{i_h} \to a_{i_1 x_1} ... a_{i_h x_h}$
2. $a_{i_1} ... a_{i_h} \to a_{i_1 x_1} ... a_{i_h x_h} c_{j_{1come}} ... c_{j_{lcome}}$

where $h, l \geq 1$, and $x_m \in \{here, out\}$ for $1 \leq m \leq h$, and the a's and c's are symbols. Call this system ECPS+. ECPS+ is still equivalent to a VASS. Thus, we have:

Corollary 2. *The following systems are equivalent: Sequential 1-membrane ECPS, sequential 1-membrane ECPS+, and VASS.*

3.3 Sequential 2-Membrane CPS

In Section 3.1, we saw that a sequential 1-membrane CPS can only define a semilinear set. However, if the system has two membranes, we can show:

Theorem 7. *A sequential 2-membrane CPS is equivalent to a VASS.*

3.4 Sequential Multi-Membrane ECPS

In Theorem 6, we saw that a sequential 1-membrane ECPS can be simulated by a VASS. This result generalizes to:

Theorem 8. *The following are equivalent: VASS, sequential 2-membrane CPS, sequential 1-membrane ECPS, sequential multi-membrane ECPS, and sequential multi-membrane ECPS+.*

We can also prove:

Theorem 9. *For any n, a multi-membrane ECPS+ operating under $\leq n$-***Parallel** *mode or n-***Parallel** *mode is equivalent to a VASS.*

4 Membrane Hierarchy

The question of whether there exists a model of P systems where the number of membranes induces an infinite hierarchy in its computational power had been open since the beginning of membrane computing five years ago. Our recent paper [10] provided a positive answer to this open problem.

Consider a restricted model of a communicating P system, called RCPS, whose environment does not contain any object initially. The system can expel objects into the environment but only expelled objects can be retrieved from the environment. Such a system is initially given an input $a_1^{i_1}...a_n^{i_n}$ (with each i_j representing the multiplicity of distinguished object a_i, $1 \leq i \leq n$) and is used as an acceptor. We showed the following results in [10]:

Theorem 10. 1. *RCPS's are equivalent to two-way multihead finite automata over bounded languages (i.e., subsets of $a_1^*...a_n^*$, for some distinct symbols $a_1, ..., a_n$).*
2. *For every r, there is an $s > r$ and a unary language L accepted by an RCPS with s membranes that cannot be accepted by an RCPS with r membranes.*

We note that the proof of the infinite hierarchy above reduces the problem (in an intricate way) to the known hierarchy of nondeterministic two-way multihead finite automata over a unary input alphabet. An interesting problem for further investigation is whether the hierarchy can be made tighter, i.e., whether the result holds for $s = r + 1$.

We also considered in [10] variants/generalizations of RCPS's, e.g, acceptors of languages; models that allow a “polynomial bounded” supply of objects in the environment initially; models with tentacles, etc. We showed that they also form an infinite hierarchy with respect to the number of membranes (or tentacles). The proof techniques can be used to obtain similar results for other restricted models of P systems, like symport/antiport systems.

5 Computational Complexity of P Systems

In [11], we showed how techniques in machine-based complexity can be used to analyze the complexity of membrane computing systems. The focus was on catalytic systems, communicating P systems, and systems with only symport/antiport rules, but the techniques are applicable to other P systems that are universal. We defined space and time complexity measures and showed hierarchies of complexity classes similar to well known results concerning Turing machines and counter machines. We also showed that the deterministic communicating P system simulating a deterministic counter machine in [19, 21] can be constructed to have a fixed number of membranes, answering positively an open question in [19, 21]. We proved that reachability of extended configurations for symport/antiport systems (as well as for catalytic systems and communicating P systems) can be decided in nondeterministic $log\ n$ space and, hence, in deterministic $log^2 n$ space or in polynomial time, improving the main result in [17]. We also proposed two equivalent systems that define languages (instead of multisets of objects): the first is a catalytic system language generator and the other is a communicating P system acceptor (or a symport/antiport system acceptor). These devices are universal and therefore can also be analyzed with respect to space and time complexity. Finally, we gave a characterization of semilinear languages in terms of a restricted form of catalytic system language generator.

6 Some Problems for Future Research

Limited parallelism in other P systems: We believe the results in Sections 2 and 3 can be shown to hold for other more general P systems (including those where membranes can be dissolved), provided the rules are not prioritized. For example, the results should apply to systems with symport/antiport rules. We plan to look at this problem.

Characterizations: We propose to investigate various classes of nonuniversal P systems and characterize their computing power in terms of well-known models of sequential and parallel computation. We plan to investigate language-theoretic properties of families of languages defined by P systems that are not universal (e.g., closure and decidable properties), find P system models that correspond to the Chomsky hierarchy, and in particular, characterize the "parallel" computing power of P systems in terms of well-known models like alternating Turing machines, circuit models, cellular automata, parallel random access machines. We will also study models of P systems that are not universal for which we can develop useful and efficient algorithms for their decision problems.

Reachability problem in cell simulation: Another important research area that has great potential applications in biology is the use of P systems for the modeling and simulation of cells. While previous work on modeling and simulation use continuous mathematics (differential equations), P systems will allow us to use discrete mathematics and algorithms. As a P system models the computation that occurs in a living cell, an important problem is to develop tools for determining reachability between configurations, i.e., how the system evolves over time. Specifically, given a P system and two configurations α and β (a configuration is the number and distribution of the different types of objects in the various membranes in the system), is β reachable from α? Unfortunately, unrestricted P systems are universal (i.e., can simulate a Turing machine), hence all nontrivial decision problems (including reachability) are undecidable. Therefore, it is important to identify special P systems that are decidable for reachability.

7 Conclusion

We showed in this paper that P systems that operate under limited parallelism are strictly weaker than systems that operate in "maximal parallelism" for two classes of systems: multi-membrane catalytic systems and multi-membrane communicating P systems. Our results on multi-membrane communicating P systems should also hold for an equivalent model with symport/antiport rules [12, 14]. We also briefly summarized our recent results concerning membrane hierarchy and computational complexity of P systems. Finally, we proposed some problems for future research.

There has been some related work on P systems operating in sequential mode. For example, sequential variants of P systems have been studied, in a different framework, in [4]. There, generalized P systems (GP-systems) were considered and were shown to be able to simulate graph controlled grammars. A comparison between parallel and sequential modes of computation in a restricted model of a P automaton was also recently investigated in [2], where it was shown that the parallel version is equivalent to a linear space-bounded nondeterministic Turing machine (NTM) and the sequential version is equivalent to a simple type of a one-way $log\ n$ space-bounded NTM.

References

1. H. G. Baker. Rabin's proof of the undecidability of the reachability set inclusion problem for vector addition systems. In *C.S.C. Memo 79, Project MAC, MIT*, 1973.
2. E. Csuhaj-Varju, O. Ibarra, and G. Vaszil. On the computational complexity of P automata. In *DNA 10*, Lecture Notes in Computer Science. Springer-Verlag, 2004 (to appear).
3. J. Esparza. Petri nets, commutative context-free grammars, and basic parallel processes. In *Proc. Fundamentals of Computer Theory*, volume 965 of *Lecture Notes in Computer Science*, pages 221–232. Springer, 1995.
4. R. Freund. Sequential P-systems. Available at *http://psystems.disco.unimib.it*, 2000.
5. R. Freund, L. Kari, M. Oswald, and P. Sosik. Computationally universal P systems without priorities: two catalysts are sufficient. Available at *http://psystems.disco.unimib.it*, 2003.
6. R. Freund and A. Paun. Membrane systems with symport/antiport rules: universality results. In *Proc. WMC-CdeA2002*, volume 2597 of *Lecture Notes in Computer Science*, pages 270–287. Springer, 2003.
7. M. H. Hack. The equality problem for vector addition systems is undecidable. In *C.S.C. Memo 121, Project MAC, MIT*, 1975.
8. J. Hopcroft and J.-J. Pansiot. On the reachability problem for 5-dimensional vector addition systems. *Theoretical Computer Science*, 8(2):135–159, 1979.
9. D.T. Huynh. Commutative grammars: The complexity of uniform word problems. *Information and Control*, 57:21–39, 1983.
10. O. H. Ibarra. The number of membranes matters. In *Proceedings of the 2003 Workshop on Membrane Computing (WMC 2003)*, Lectures Notes in Computer Science, to appear.
11. O. H. Ibarra. On the computational complexity of membrane systems. *Theoretical Computer Science*, to appear.
12. C. Martin-Vide, A. Paun, and Gh. Paun. On the power of P systems with symport rules. *Journal of Universal Computer Science*, 8(2):317–331, 2002.
13. E. Mayr. Persistence of vector replacement systems is decidable. *Acta Informat.*, 15:309–318, 1981.
14. A. Paun and Gh. Paun. The power of communication: P systems with symport/antiport. *New Generation Computing*, 20(3):295–306, 2002.
15. Gh. Paun. Computing with membranes. *Journal of Computer and System Sciences*, 61(1):108–143, 2000.
16. Gh. Paun. *Membrane Computing: An Introduction*. Springer-Verlag, 2002.
17. Gh. Paun, M. Perez-Jimenez, and F. Sancho-Caparrini. On the reachability problem for P systems with symport/antiport. *submitted*, 2002.
18. Gh. Paun and G. Rozenberg. A guide to membrane computing. *Theoretical Computer Science*, 287(1):73–100, 2002.
19. P. Sosik. P systems versus register machines: two universality proofs. In *Pre-Proceedings of Workshop on Membrane Computing (WMC-CdeA2002), Curtea de Arges, Romania*, pages 371–382, 2002.
20. P. Sosik and R. Freund. P systems without priorities are computationally universal. In *WMC-CdeA2002*, volume 2597 of *Lecture Notes in Computer Science*, pages 400–409. Springer, 2003.
21. P. Sosik and J. Matysek. Membrane computing: when communication is enough. In *Unconventional Models of Computation 2002*, volume 2509 of *Lecture Notes in Computer Science*, pages 264–275. Springer, 2002.
22. J. van Leeuwen. A partial solution to the reachability problem for vector addition systems. In *Proceedings of STOC'74*, pages 303–309.

Cellular Meta-Programming

Gabriel Ciobanu

Romanian Academy, Institute of Computer Science, Iaşi
gabriel@iit.tuiasi.ro

Abstract. Adaptable executions inspired by the cell behaviour can be described by a cellular meta-programming paradigm. The cell adaptability and meta-programming are related to the notions of behavioural reflection, which allows a program to modify, even at run-time, its own code as well as the semantics of its own programming language. We present the cellular meta-programming helping on membrane systems and a specification language based on rewriting which allows meta-level strategies and use of reflection.

1 Systems Biology

Systems biology represents a new cross-disciplinary approach in biology which has only recently been made possible by advances in computer science and technology. Unlike traditional biologists studying individual genes, proteins or small subsystems in isolation, systems biology embraces the view that the whole system must be analyzed in order to understand the complex interaction of all levels of biological information. Systems biology requires global specifications and tools both to define the elements of the system, and to follow the elements behaviour as the system carries out its functions. Finally, the systems approach requires mathematical models and methods able to describe the nature of the whole system, and its properties. Therefore systems biology is a new holistic view of molecular biology. As it is mentioned in [8], it involves the application of experimental, theoretical, and modelling techniques to the study of biological organisms at all levels, from the molecular, through the cellular, to the behavioural. Its aim and challenge is to identify the principles that lead to the actual combination of molecular mechanisms. Adding new abstractions, discrete models and methods able to help our understanding of the biological phenomena, systems biology may provide predictive power, useful classifications, new paradigms in computing and new perspectives on the dynamics of various biological systems.

In this paper we present cellular meta-programming, a computing paradigm inspired by the dynamic nature of the cell behaviour, described with the help of membrane systems representing abstract models inspired by the compartments of a cell. The root of this approach is given by the high adaptability and flexibility of the cell behaviour. Identifying the principles that govern the design and function of this adaptability is a central goal of our research. The adaptation of cells to the changing environment requires sophisticated processing mediated by interacting genes and proteins. In computing terms, we say that a cell is able to adapt its execution according to various developmental and environmental stimuli, causing corresponding changes in its behaviour. We refer mainly to adaptability at the software level.

Adaptable executions are generated in computer science by meta-programming. Meta-programming is the act of writing meta-programs, and a meta-program is a program that manipulates itself as its data, allowing execution modification. Meta-programming is related to the notions of reflection. In the programming languages, reflection is defined as the ability of a program to manipulate the encoding of the state of the program during its own execution. The mechanism for encoding execution states (as data) is called reification. The reflective mechanisms are both structural and behavioural. Structural reflection is the ability to work with the structures and processes of a programming system within the programming system itself. This form of reflection is easier to implement, and languages as Lisp, Smalltalk, and Java have structural reflection mechanisms. However, the cell adaptability is close to the behavioural reflection which allows a program to modify, even at run-time, its own code as well as the semantics and the implementation of its own programming language. Our main interest is in behaviour reflection.

The cell is able to modify its activity according to its foregoing processes, to observe and change its own code even at run-time. This behaviour is more than the reification process by which a program which is at run-time is used as a representation (data structures, procedures) expressed in the language itself, and made available to the program as ordinary data. The cells programming languages allow to step into the meta-level, where implementation of the language is explicitly available. This capability enables various customizations including new structures, changing the behaviour, going up and down in a tower of meta-levels by using a reflection mechanism.

We use a rather mathematical language (called Maude) which is able to provide executable specification, and an abstract model of membranes inspired by the compartments of a cell (called P systems). In order to present the cellular meta-programming paradigm, we provide the executable specifications of P systems in Maude, a software system supporting reflection. Despite the theoretical limits, Meseguer and Clavel have defined a general theory of reflection for Maude in [9]. They propose meta-logical axioms for reflection, as well as general axioms for computational strategies in rewriting logic.

2 Cellular Meta-Programming over Membranes

Membrane systems represent a new abstract model of parallel and distributed computing inspired by cell compartments and molecular membranes [10, 11]. A cell is divided in various compartments, each compartment with a different task, and all of them working simultaneously to accomplish a more general task of the whole system. The membranes of a P system determine regions where objects and evolution rules can be placed. The objects evolve according to the rules associated with each region, and the regions cooperate in order to maintain the proper behaviour of the whole system. P systems provide a nice abstraction for parallel systems, and a suitable framework for distributed and parallel algorithms [2].

This section presents briefly an executable specifications for membrane systems. More details are presented in [1]. We emphasize the cellular meta-programming and reflection aspects, providing in the same time a distinctive feature of the membrane computing.
Formally, a P system is a structure $\Pi = (O, \mu, w_1, \ldots, w_m, R_1, \ldots, R_m, i_o)$, where:

(i) O is an alphabet of objects;
(ii) μ is a membrane structure consisting of labelled membranes;
(iii) w_i are multisets over O associated with the regions defined by μ;
(iv) R_i are finite sets of evolution rules over O associated with the membranes, of typical form $ab \to a(c, in_2)(c, out)$;
(v) i_0 is either a number between 1 and m specifying the *output* membrane of Π, or it is equal to 0 indicating that the output is the outer region.

We prefer to express a membrane as a structure $M = (R_M, w_M)$, and its evolution rules as rewriting rules. We consider the following maximal parallel application of rules: in a transition step, the rules of each membrane are used against its resources such that no more rules can be applied. Considering an *elementary membrane* $M = (R_M, w_M)$, where R_M is the finite set of evolution rules and w_M is the initial multiset, a computation step transition is defined as a rewriting rule by

$$\frac{x_1 \to y_1, \ldots, x_n \to y_n \in R_M, z \text{ is } R_M\text{-irreducible}}{x_1 \ldots x_n \Rightarrow y_1 \ldots y_n z} \tag{1}$$

z is R_M-irreducible whenever there does not exist rules in R_M applicable to z. A *composite membrane*, that is a membrane with other membranes $M_1, \ldots, M_k$ inside it, is denoted by $(M_1, \ldots, M_k, R_M, init)$, where each $M_i (1 \leq i \leq k)$ is an elementary or a composite membrane. R_M represents the finite set of evolution rules of M, and *init* is its initial configuration of form $(w, (w_1, \ldots, w_k))$, where w_i is the multiset associated with the membrane M_i. A computational step of a composite membrane is defined as a rewriting rule by

$$\frac{w \Rightarrow w', w_1 \Rightarrow w'_1, \ldots, w_n \Rightarrow w'_n}{(w, (w_1, \ldots, w_k)) \Rightarrow (w', (w'_1, \ldots, w'_k))} \tag{2}$$

In this way, the objects of the membranes are the subject of local evolution rules that evolve simultaneously. A sequence of computation steps represents a computation. A computation is successful if this sequence is finite, namely there is no rule applicable to the objects present in the last configuration. In a final configuration, the result of a successful computation is the total number of objects present in the membrane considered as the output membrane. We simplify this procedure; no internal membrane is specified as an output membrane, and so the result is given by the number of objects in the skin membrane.

Maude is essentially a mathematical language. The OBJ theory and languages [7] have influenced the Maude design and philosophy. A Maude program is a logical theory, and a Maude computation is a logical deduction using the axioms specified in the program. The foundations of Maude is given by membership equational logic and rewriting logic. A rewriting specification $\mathcal{R}$ is a 4-tuple $\mathcal{R}=(\Sigma, E, L, R)$ where

(Σ, E) is a rewriting logic signature, L is a set whose elements are called labels, and R is a set of labelled rewriting rules (sentences) written as $r : [t(\overline{x})]_E \to [t'(\overline{x})]_E$. The inference rules of rewriting logic allow to deduce general (concurrent) transitions which are possible in a system satisfying $\mathcal{R}$. We say that $\mathcal{R}$ *entails the sentence* $[t] \to [t']$ and write $\mathcal{R} \vdash [t] \to [t']$ iff $[t] \to [t']$ can be obtained by finite application of its inference rules. The general theory of the rewriting logic allows conditional sentences and conditional rewriting rules. The interested reader is invited to read [4].
The basic programming statements of Maude are equations, membership assertions, and rules. A Maude program containing only equations and membership assertions is called a functional module. The equations are used as rules (equational rewriting), and the replacement of equals for equals is performed only from left to right. A Maude program containing both equations and rules is called a system module. Rules are not equations, they are local transition rules in a possibly concurrent system. Unlike for equations, there is no assumption that all rewriting sequences will lead to the same final result, and for some systems there may not be any final states.

The rewriting performed for membranes is a multiset rewriting. In Maude this is specified in the equational part of the program (system module) by declaring that the multiset union operator satisfies the associativity and commutativity equations, and has also an identity. This is done simply by using attributes, and this information is used to generate a multiset matching algorithm. Further expressiveness is gained by various features as equational pattern matching, user-definable syntax and data, generic types and modules, and reflection.

We emphasize the *evaluation strategies* and *reflection property*. Evaluation strategies control the positions in which equations can be applied, giving the user the possibility of indicating which arguments to evaluate before simplifying a given operator with the equations. Reflection allows a complete control of the rewriting (execution) using the rewriting rules in the theory. Reflective computations allow the link between meta-level and the object level, whenever possible.

Rewriting logic is *reflective*, i.e. there is a finitely presented *universal rewriting specification* $\mathcal{U}$ such that for any finitely presented rewriting specification $\mathcal{R}$ (including $\mathcal{U}$ itself), we have the following equivalence:

$$\mathcal{R} \vdash [t] \to [t'] \;\text{ iff }\; \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \to \langle \overline{\mathcal{R}}, \overline{t'} \rangle,$$

where $\overline{\mathcal{R}}$ and $\overline{t}$ are terms representing $\mathcal{R}$ and t as data elements of $\mathcal{U}$. Since $\mathcal{U}$ is representable in itself, it is possible to achieve a "reflective tower" with an arbitrary number of reflection levels:

$$\mathcal{R} \vdash [t] \to [t'] \text{ iff } \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \to \langle \overline{\mathcal{R}}, \overline{t'} \rangle \text{ iff } \mathcal{U} \vdash \langle \overline{\mathcal{U}}, \overline{\langle \overline{\mathcal{R}}, \overline{t} \rangle} \rangle \to \langle \overline{\mathcal{U}}, \overline{\langle \overline{\mathcal{R}}, \overline{t'} \rangle} \rangle \ldots$$

This interesting and powerful concept is supported by Maude through a built-in module called `META-LEVEL`. This module has sorts `Term` and `Module` such that the representation $\overline{t}$ of a term t is of sort `Term` and the representation $\overline{SP}$ of a specification SP is of sort `Module`. There are also functions like `metaReduce`$(\overline{SP}, \overline{t})$ which returns the representation of the reduced form of a term t using the equations in the module SP.

META-LEVEL module can be extended by the user to specify strategies of controlling the rewriting process. We use META-LEVEL in order to define the "maximal parallel rewriting" strategy. In fact the meta-level is needed for two main reasons: to locate the set of rules corresponding to a certain membrane in the structured Maude specification of a composite P system, and to describe the maximal parallel application of the located rules as a rewriting strategy.

The META-LEVEL module is used to provide a clear algorithmic description given by maxParRew of the rather awkward and ambiguous "nondeterministic and maximal parallel" applications of evolution rules in the P systems. Using maxParRew as a transition step between meta-level configurations, we then provide an operational semantics of the P systems. Using the power given by the tower of reflection levels in Maude, we define operations over modules and strategies to guide the deduction process. Finally we can use a meta-metalevel to analyze and verify the properties of the P systems [1]. This conceptual description of the P systems based on meta-programming capabilities given by reflection is called cellular meta-programming, and it could become a useful paradigm for further investigations in system biology.

3 Membrane Systems Specification in Maude

Each P system Π is naturally represented as a collection of Maude system modules such that each membrane is represented by a corresponding Maude system module. The sort Obj is for object names, and its subsort Output is for results. We add a sort Soup for the multisets of objects, and a sort Config for the states of a P system. An expression of the form $\langle M \mid S \rangle$ represents a configuration corresponding to an elementary membrane M with its multiset S, and an expression of the form $\langle M \mid S; C_1, \ldots, C_n \rangle$ represents a configuration corresponding to a composite membrane M in state S and with the component i having the configuration C_i. The Maude semantics of the module M is not the same with the P system semantics. Therefore we must associate with M the right semantics based on the maximal parallel rewrite relation. We use the facilities provided by reflection in Maude, defining this semantics at the meta-level.

For the elementary membranes, a computation step between configurations is defined as:

$$\frac{S \Rightarrow S'}{\langle M \mid S \rangle \Rightarrow \langle M \mid S' \rangle} \tag{3}$$

where $S \Rightarrow S'$ is defined in (1). $S \Rightarrow S'$ is not the ordinary rewriting defined by M, but they are strongly related:

$$S \Rightarrow S' \text{ iff } S \xrightarrow{+}_{R_M} S' \text{ s.t. } \mathit{maxParCons}(R_M, S, S')$$

where $\xrightarrow{+}_{R_M}$ is the ordinary rewriting defined by R_M, and $\mathit{maxParCons}(R_M, S, S')$ represents the constraints defining the maximal parallel rewriting strategy over R_M. More precisely, we have:

1. if $S = S'$, then $\mathit{maxParCons}(R_M, S, S)$ holds iff S is R_M-irreducible;

2. if $S \neq S'$, then $maxParCons(R_M, S, S')$ holds iff there exists $S_1, S'_1, \ell \to r \in R_M$ such that $S = \ell\, S_1$, $S' = r\, S'_1$, and $maxParCons(R_M, S_1, S'_1)$.

Since *maxParCons* has the set of rules of the module M as parameter, it follows that it can be decided only at meta-level. The transition between configurations for composite membrane is defined as:

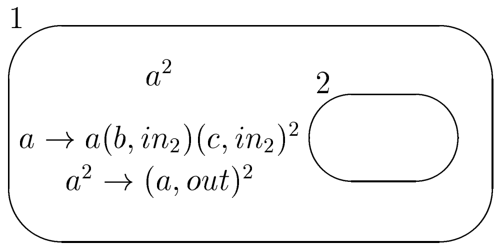
$$\frac{S \Rightarrow S', C_1 \Rightarrow C'_1, \ldots, C_k \Rightarrow C'_k}{\langle M \mid S; C_1, \ldots, C_k\rangle \Rightarrow \langle M \mid S'; C'_1, \ldots, C'_k\rangle} \tag{4}$$

A computation is a sequence of transitions steps $C_0 \Rightarrow C_1 \Rightarrow C_2 \Rightarrow \ldots \Rightarrow C_n \Rightarrow \ldots$, where C_0 is the initial configuration. The result of a successful computation is extracted from the final configuration; for instance, the result could be the total number of objects present in the skin membrane.

Example: We present a simple example of membrane system, and then describe and execute its Maude specification. We consider a P system generating symbols b and c with the properties that the number of c's is double of the number of b's, and the total number of b's and c's is a multiple of 6.

$$\begin{aligned}
&\Pi_1 = (O, \mu, w_1, w_2, R_1, R_2, i_o),\\
&\quad O = \{a, b, c\},\\
&\quad \mu = [_1[_2\ ]_2]_1,\\
&\quad w_1 = a^2,\\
&\quad w_2 = \lambda,\\
&\quad R_1 = \{a \to a(b, in_2)(c, in_2)^2,\ a^2 \to (a, out)^2\},\\
&\quad R_2 = \emptyset,\\
&\quad i_o = 2.
\end{aligned}$$

The initial configuration is:



We present here only some important steps of the specification; more details can be found in [1]. Each membrane is specified in Maude by an independent system module. Actually, we ignore the membrane labelled by 2, and consider Π_1 consisting only of the skin:

```
(mod SKIN is
  inc CONFIG(OBJ-TO-ABC) .
  op init : -> Soup .
  eq init = a a .
  rl ['SKIN] : a => a b c c .
  rl ['SKIN] : a a => empty .
endm)
```

The Maude specification of a P system is a system module importing the modules corresponding to the component membranes, and defining the initial configuration. The structure of the system is specified in the initial configuration. The module describing Π_1 is:

```
(mod PSYS is
  inc SKIN .
  op initConf : -> Config .
  eq initConf = < 'SKIN | init > .
endm)
```

The rewriting rules defining the computation of P systems are included in a Maude system module called `COMPS`. For instance, some rules are implemented in Maude at the meta-level as:

```
crl [ps1] : maxParRewS(R, S) =>
    maxParRewS(R, (rl  X =>  Y [label(Q)] .), MP, S)
    if (R1 rl X => Y [label(Q)] . R2) := R /\
       MP := metaXmatch(PSYS, getTerm(metaReduce(PSYS, X)),
             getTerm(metaReduce(PSYS,S)), nil, 0, unbounded, 0) /\
       MP :: MatchPair .
crl [ps4] : maxParRew('<_|_>[M , S]) =>
    '<_|_>[M, maxParRewS(getQRls(getRls(PSYS), M), S)]
    if sameKind(PSYS, getType(metaReduce(PSYS, S)), 'Soup) .
```

We can use various Maude commands in order to make experiments with the P system specification. For instance, we use the command `rew` to see the result of maximal parallel rewritings:

```
Maude> select COMPS .
Maude> (down PSYS : rew  rwf(getTerm(metaReduce(up(PSYS),
                                   up(PSYS, initConf)))) .)
rewrites: 4784 in 130ms cpu (140ms real) (36800 rewrites/second)
result Config :
  < 'SKIN | a a b b b b b b c c c c c c c c c c c c >
Maude>
```

The `rew` command with a limited number of steps, is not useful in our case because the number of rewriting steps in Maude is not the same with the number of computation steps of P systems. Therefore, the rewriting process is restricted by the configuration size:

```
  crl rwf(X) => rwf(maxParRew(X)) if (X :: Term) /\ (#(X) < 20) .
  crl rwf(X) => X if #(X) >= 20 .
```

We consider a module `METACOMPS` defining a function `out` which removes the non-output objects, and a function `#()` which counts the occurrences of an object into a configuration. The function `out` simulates somehow the membrane labelled by 2. Since these functions are applied to configurations obtained with `metaRewrite` command, module `METACOMPS` is defined at the meta-metalevel.

```
Maude> (select METACOMPS .)
Maude> (down PSYS : down COMPS : red getTerm(metaRewrite(up(COMPS),
                               up(COMPS, rwf(getTerm(metaReduce(up(PSYS),
                               up(PSYS, initConf))))), 100)) .)
rewrites: 26614 in 150ms cpu (140ms real) (177426 rewrites/second)
result Config :
  < 'SKIN | a a b b b b b b c c c c c c c c c c c c >
```

The command `down` and the function `up` are used to move between two successive levels of the reflection tower. For instance, `down COMPS :` interprets the result returned by `red` in the module `COMPS`. The function call `up(PSYS, initConf)` returns the representation at the meta-level of the term `initConf` defined in `PSYS`. If we wish to investigate the properties of the result, then we may proceed as follows:

```
(mod PROOF is
  inc METACOMPS .
  op ql : -> QidList .
  eq ql = out(...getTerm(metaRewrite(up(COMPS),
          up(COMPS,getTerm(metaReduce(up(PSYS),up(PSYS,initConf)))),100)))) .
endm)
```

We can check now that the number of objects `c` is double of the number of objects `b`, and the total number of b's and c's is a multiple of 6:

```
Maude> (red #(ql, ''c) == 2 * #(ql, ''b) .)
rewrites: 322 in 230ms cpu (230ms real) (1400 rewrites/second)
reduce in PROOF :
  #(ql,''c)== 2 * #(ql,''b)
result Bool :
  true
Maude> (red (#(ql, ''c) + #(ql, ''b)) rem 6 .)
rewrites: 326 in 10ms cpu (10ms real) (32600 rewrites/second)
reduce in PROOF :
  (#(ql,''c)+ #(ql,''b))rem 6
result Zero :
  0
```

Here we check certain properties for a specific configuration. Some properties can be checked for all the configurations, using the temporal formulas and a model checker implemented in Maude. Maude has a collection of formal tools supporting different forms of logical reasoning to verify program properties, including a model checker to verify temporal properties of finite-state system modules [6]. This model checker provides a useful tool to detect subtle errors, and to verify some desired temporal properties.

4 Conclusion

Starting from the cells ability to react and change their behaviour at run-time, we translate this adaptability in a meta-programming feature of a P systems implementation based on executable specifications.

We view the cell as a complex system coordinating various membrane working in parallel. Adaptable executions inspired by the cell behaviour can be described by a cellular meta-programming paradigm. The cell adaptability and meta-programming are related to the notions of behavioural reflection. We present the cellular meta-programming with the help of membrane systems and a reflective specification language based on rewriting. The approach exploits the reflection property of the rewriting logic, property which provide a meta-programming abstraction. In this way, the abstract mechanism of reflection could describe the biological entities ability to react and change their behaviour according to various developmental and environmental stimuli. In this paper we emphasize the cellular meta-programming and reflection aspects, providing in the same time a distinctive feature of the membrane computing.

Inspired by the cell compartments and their parallel executions, we present in [3] a parallel implementation of the membrane systems on a cluster of computers. Membrane systems also represent a suitable framework for distributed and parallel algorithms [2]. Further research will investigate how to integrate these aspects with the cellular meta-programming paradigm, trying to harmonize theory and practice, fostering fertilization between biological principles and computation paradigms.

References

1. O.Andrei, G.Ciobanu, D.Lucanu. Rewriting P Systems in Maude. In G.Mauri, Gh.Păun, C.Zandron (Eds.): *Pre-Proceedings WMC5*, Milano, 104-118, 2004.
2. G. Ciobanu. Distributed algorithms over communicating membrane systems. BioSystems, vol.70(2), 123-133, Elsevier, 2003.
3. G. Ciobanu, W. Guo. P Systems Running on a Cluster of Computers. In *Membrane Computing*, LNCS vol.2933, Springer, 123-139, 2004.
4. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, J.F. Quesada. Maude: Specification and Programming in Rewriting Logic. *Theoretical Computer Science*, vol.285(2), 187-243, 2002.
5. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C .Talcott. *Maude Manual* (Version 2.1). http://maude.cs.uiuc.edu, 2004.
6. S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL Model Checker and Its Implementation. In T.Ball, S.K.Rajamani (Eds.): *Model Checking Software: 10th SPIN Workshop*, LNCS vol.2648, Springer, 230-234, 2003.
7. J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, J.P. Jouannaud. Introducing OBJ. In *Software Engineering with OBJ*, 3-167, Kluwer, 2000.
8. H. Kitano. Computational Systems Biology. Nature vol.420, 206-210, 2002.
9. J. Meseguer, M. Clavel. Axiomatizing Reflective Logics and Languages. In G.Kiczales (Ed.): *Reflection'96*, 263-288. Xerox PARC, 1996.
10. Gh. Păun. Computing with membranes. *Journal of Computer and System Sciences* vol.61, 108-143, 2000.
11. Gh. Păun. *Computing with Membranes: An Introduction*, Springer, 2002.

From Quantum Computing to Quantum Programming

Philippe Jorrand

CNRS - Leibniz Laboratory
46, avenue Félix Viallet
38000 Grenoble, France
`Philippe.Jorrand@imag.fr`

Extended Abstract

1 From Quantum Physics to Quantum Computing, ...

Information is physical: the laws which govern its encoding, processing and communication are bound by those of its unavoidably physical incarnation. In today's informatics, information obeys the laws of classical Newtonian physics: this statement holds all the way from commercial computers down to (up to?) Turing machines and lambda-calculus. Today's computation is classical.

The driving force of research in quantum computation is that of looking for the consequences of having information encoding, processing and communication based upon the laws of another kind of physics, namely quantum physics, i.e. the ultimate knowledge that we have, today, of the foreign world of elementary particles, as described by quantum mechanics.

Quantum mechanics, which is the mathematical formulation of the laws of quantum physics, relies on four postulates: (*i*) the state of a quantum system (i.e. a particle, or a collection of particles) is a unit element of a Hilbert space, that is a vector of length 1 in a d-dimensional complex vector space; (*ii*) the evolution of the state of a closed quantum system (i.e. not interacting with its -classical- environment) is deterministic, linear, reversible and characterized by a unitary operator, that is by a $d \times d$ rotation matrix applied to the state vector; (*iii*) the measurement of a quantum system (i.e. the observation of a quantum system by its -classical- environment) irreversibly modifies the state of the system by performing a projection of the state vector onto a probabilistically chosen subspace of the Hilbert space, with renormalization of the resulting vector, and returns a value (e.g. an integer) to the classical world, which just tells which subspace was chosen; and (*iv*) the state space of a quantum system composed of several quantum subsystems is the tensor product of the state spaces of its components (given two vector spaces P and Q of dimensions p and q respectively, their tensor product is a vector space of dimension $p \times q$).

The question is then: how to take advantage of these postulates to the benefits of computation?

The most widely developed approach to quantum computation exploits all four postulates in a rather straightforward manner. The elementary physical carrier of information is a qubit (quantum bit), i.e. a quantum system (electron, photon, ion, ...) with a 2-dimensional state space (postulate *i*); the state of a n-qubit register lives in a 2^n-dimensional Hilbert space, the tensor product of n 2-dimensional Hilbert spaces (postulate *iv*). Then, by imitating in the quantum world the most traditional organization of classical computation, quantum computations are considered as comprising three steps in sequence: first, preparation of the initial state of a quantum register (postulate *iii* can be used for that, possibly with postulate *ii*); second, computation, by means of deterministic unitary transformations of the register state (postulate *ii*); and third, output of a result by probabilistic measurement of all or part of the register (postulate *iii*).

More concretely, from a computational point of view, these postulates provide the elementary quantum ingredients which are at the basis of quantum algorithm design:

Superposition: at any given moment, the state of quantum register of n qubits is a vector in a 2^n-dimensional vector space, i.e. a vector with at most 2^n non zero components, one for each of the 2^n different values on n bits: the basis of this vector space comprises the 2^n vectors $|i>$, for i in $\{0,1\}^n$ ($|i>$ is Dirac's notation for vectors denoting quantum states). This fact is exploited computationally by considering that this register can actually contain (a superposition of) all the 2^n different values on n bits, whereas a classical register of n bits may contain only one of these values at any given moment.

Quantum parallelism and deterministic computation: let f be a function from $\{0,1\}^n$ to $\{0,1\}^n$ and x be a quantum register of n qubits initialized in a superposition of all values in $\{0,1\}^n$ (this initialization can be done in one very simple step). Then, computing $f(x)$ is achieved by a deterministic, linear and unitary operation on the state of x: because of linearity, a single application of this operation produces all 2^n values of f in one computation step. Performing this operation for any, possibly non linear f while obeying the linearity and unitarity laws of the quantum world, requires a register of $2n$ qubits formed of the register x, augmented with a register y of n qubits initialized in the basis state $|0>$: before the computation of f, this larger register contains a superposition of all pairs $|i,0>$ for i in $\{0,1\}^n$ and, after the computation of f, it contains a superposition of all pairs $|i,f(i)>$ for i in $\{0,1\}^n$.

Probabilistic measurement and output of a result: after f has been computed, all its values $f(i)$, for i in $\{0,1\}^n$, are superposed in the y part of the register of $2n$ qubits, each of these values facing (in the pair $|i,f(i)>$) their corresponding i still stored within the unchanged superposition contained in the x part of that register. Observing the contents of y will return only one value, j, among the possible values of f. This value is chosen with a probability which depends on f since, e.g. if $f(i)=j$ for several distinct values of i, the probability of obtaining j as a result will be higher than that of obtaining k if $f(i)=k$ for only one value of i. This measurement also causes the superposition in y to be projected onto the 1-dimensional subspace corresponding to the basis state $|j>$, i.e. the state of the y part collapses to $|j>$, which implies that all other values of f which were previously in y are irreversibly lost.

Interference: the results of the 2^n parallel computations of f over its domain of definition can be made to interfere with each other. Substractive interference will lower the probability of observing some of these value in y, whereas additive interference will increase the probability of observing other values and bring it closer to 1.

Entangled states: measuring y after the computation of f is in fact measuring only n qubits (the y part) among the $2n$ qubits of a register. The state of this larger register is a superposition of all pairs $|i,f(i)>$ for i in $\{0,1\}^n$ (e.g., in this superposition, there is no pair like $|2,f(3)>$): this superposition is not a free cross-product of the domain $\{0,1\}^n$ of f by its image in $\{0,1\}^n$, i.e. there is a strong correlation between the contents of the x and y parts of the register. As a consequence, if measuring the y part returns a value j, with the state of that part collapsing to the basis state $|j>$, the state of the larger register will itself collapse to a superposition of all remaining pairs $|i,j>$ such that $f(i)=j$. This means that, in addition to producing a value j, the measurement of the y part also causes the state of the x part to collapse to a superposition of all elements of the $f^{-1}(j)$ subset of the domain of f. This correlation is called entanglement: in quantum physics, the state of a system composed of n sub-systems is not, in general, simply reducible to an n-tuple of the states of the components of that system. Entanglement has no equivalent in classical physics and it constitutes the most powerful resource for quantum information processing.

No cloning: a direct consequence of the linearity of all operations that can be applied to quantum states (a two line trivial proof shows it) is that the state of a qubit a (this state is in general an arbitrary superposition, i.e. a vector made of a linear combination of the two basis state vectors $|0>$ and $|1>$), cannot be duplicated and made the state of another qubit b, unless the state of a is simply either $|0>$ or $|1>$ (i.e. not an arbitrary superposition). This is true of the state of all quantum systems, including of course registers of n qubits used by quantum computations.

With these basic ingredients and their peculiarities at hand for building computations, what are then the assembling tools and the rules of the game of quantum algorithmics and their consequences for programming?

2 ... to Quantum Algorithms, ...

Richard Feynman launched in 1982 the idea that computation based upon quantum physics would be exponentially more efficient than based upon classical physics. Then, after the pioneering insight of David Deutsch in the mid eighties, who showed, by means of a quantum Turing machine, that quantum computing could indeed not, in general, be simulated in polynomial time by classical computing, it was ten years before the potential power of quantum computing was demonstrated on actual computational problems.

The first major breakthrough was by Peter Shor: in 1994, he published a quantum algorithm operating in polynomial time ($O(\log^3 N)$) for factoring an integer N, whereas the best classical algorithm is exponential. Two years later, Lov Grover published a quantum algorithm for searching an unordered database of size N, which realizes a quadratic acceleration (it operates in O($N^{1/2}$)) when compared with classical algorithms for the same problem (in O(N)). Shor's algorithm relies on a known reduction of the problem of factoring to that of finding the order of a group, or the period of a function: then, since order finding can be achieved by a Fourier Transform, the key is a Quantum Fourier Transform designed by Shor, which is indeed exponentially more efficient than FFT, thanks to quantum parallelism, entanglement and tensor product. Grover's algorithm relies upon a very subtle use of interference, now known as amplitude amplification, which performs a stepwise increase of the probability of measuring a relevant item in the database, and which brings this probability very close to one after $N^{1/2}$ steps.

Another major result, by Charles Bennet and others in 1993, was the design of theoretical principles leading to a quantum teleportation protocol, which takes advantage of entanglement and of probabilistic measurement: the state of a quantum system *a* (e.g. a qubit) localized at *A*'s place can be assigned, after having been measured, thus destroyed, to another quantum system *b* (e.g. another qubit), localized at *B*'s place, without the state of *a* being known neither by *A* nor by *B*, and without neither *a*, *b* nor any other quantum system being moved along a trajectory between *A* and *B*. It is important to notice that this is not in contradiction with no cloning: there is still only one instance of the teleported state, whereas cloning would mean that there coexist one original and one copy.

Since then, these results have been generalized and extended to related classes of problems. Shor's algorithm solves an instance of the hidden subgroup problem for abelian groups and a few extensions to non-abelian cases have been designed. In addition to Fourier Transform, order finding and amplitude amplification, other candidates to the status of higher level building blocks for quantum algorithmics have emerged, such as quantum random walks on graphs. Principles for distributed quantum computing have also been studied and successfully applied to a few classes of problems, etc. Very recently, on the basis of amplitude amplification, quadratic and other quantum speedups have been found for several problems on graphs, such as connectivity, minimum spanning tree and single source shortest paths.

Teleportation also has been generalized. The measurement used in its original formulation was such that the state eventually obtained for *b* was the same as the state initially held by *a* (up to a correcting operation which still had to be applied, depending on the probabilistic outcome of that measurement). By changing the way the measurement is done (in fact, by appropriately rotating the basis upon which the measurement of *a* will project the state of *a*), it has been found that the state teleported to *b* could be not the state initially held by *a*, but that state to which a rotation, i.e. a unitary operation has been applied. In other words, entanglement and measurement, i.e. the resources needed by teleportation, can be used to simulate computations by unitary tranformations. This has given rise to a whole new direction of research in quantum computation, namely measurement-based quantum computation.

There is an obvious and ever present invariant in all these different ways of organizing quantum computations and quantum algorithms. Quantum computations operate in the quantum world, which is a foreign and unknowable world. No one in the classical world will ever know what the superposition state of an arbitrary qubit is, the only information one can get is 0 or 1, through measurement, i.e. the classical outcome of a probabilistic projection of the qubit state vector onto |0> or |1>: if one gets |0>, the only actual information which is provided about the state before measurement is that it was not |1>, because |0> and |1> are orthogonal vectors. Then, for the results of quantum computations to be useful in any way, there is an intrinsic necessity of cooperation and communication controlled by the classical world. All quantum algorithms, either based upon unitary transformations or upon measurements, if they are of any relevance, eventually end up in a final quantum state which hides, among its superposed basic states, a desired result. Such a result is asked for upon request by the classical world, which decides at that point to perform a measurement on part or all of the quantum register used by the computation. But measurement is probabilistic: its outcome may be a desired result, but it may well be something else. For example, Grover's algorithm ends up in a state where desired results have a probability close to 1 to be obtained, but other, unwanted results may also come out from the final measurement, although with a much lower probability.

The whole game of quantum algorithmics is thus to massage the state of the quantum register so that, in the end, desired results have a high probability to be obtained, while doing that at the minimum possible cost, i.e. minimal number of operations applied (time) and of qubits used (space). This is achieved through interferences (by means of appropriate unitary operations), through the establishment of entangled states and through measurements in appropriate bases. But this is not the end: once a measurement outcome is obtained by the classical world, it must be checked, by the classical world, for its validity. If the result satisfies the required conditions to be correct, termination is decided by the classical world. If it does not, the classical world decides to start the quantum part of the computation all over. For example, in the case of Grover's algorithm, if the element of the database produced by the measurement is not correct, the whole quantum search by amplitude amplification is started again by the classical world.

In general, algorithms will not contain one, but several quantum parts embedded within classical control structures like conditions, iterations, recursions. Measurement is not the only channel through which the classical and quantum worlds interact, there is also the initialization of quantum registers to a state chosen by the classical world (notice that such initializations can only be to one among the basis states, since they are the only quantum states which correspond, one to one, to values expressible by the classical world). A quantum part of an algorithm may also, under the control of the classical world, send one of its qubits to another quantum part (but not the state of that qubits, because of no cloning): this quantum to quantum communication is especially useful for quantum secure communication protocols, which are a family of distributed quantum algorithms of very high practical relevance, in a not too far future, among the foreseeable applications of quantum information processing.

This means that not only the peculiarities of the basic quantum ingredients for computing have to be taken into account in the design of languages for the formal description of quantum algorithms (and quantum protocols), but also the necessity of embedding quantum computations within classical computations, of having both worlds communicate and cooperate, of having classical and quantum parts be arbitrarily intermixed, under the control of the classical side, within the same program.

3 ... and to Quantum Programming.

Quantum computing is still in its infancy, but quantum programming is even much younger. Quantum computing is on its way to becoming an established discipline within computer science, much like, in a symmetric and very promising manner, quantum information theory is becoming a part of quantum physics. Since the not so old birth of quantum computing, the most important efforts have been invested in the search for new quantum algorithms that would show evidence of significant drops in complexity compared with classical algorithms. Obtaining new and convincing results in this area is clearly a crucial issue for making progress in quantum computing. This research has been, as could be expected, largely focusing on complexity related questions, and relying on approaches and techniques provided by complexity theory.

However, the much longer experience from classical computer science tells that the study of complexity issues is not the only source of inspiration toward the creation, design and analysis of new algorithms. There are other roads, which run across the lands of language design and semantics. A few projects in this area have recently started, following these roads. Three quantum programming language styles are under study: imperative, parallel and distributed, and functional. There are also a number of very specific issues in the domain of semantic frameworks for quantum programming languages.

The sequential and imperative programming paradigm, upon which all major quantum algorithmic breakthroughs have relied, is still widely accepted as “the” way in which quantum + classical computations are organized and should be designed. However, before any language following that style was designed, and even today, the quantum parts of algorithms are described by drawing quantum gate arrays, which are to quantum computing what logical gate circuits are to classical computing. This is of course very cumbersome and far from useful for proving properties of programs. This is why some imperative languages for quantum + classical programming have been design first.

The most representative quantum imperative programming language is QCL (Quantum Computing Language), a C flavoured language designed by B. Ömer at the University of Vienna. Another one, qGCL (Quantum Guarded Command Language) was due to P. Zuliani at Oxford University, with the interesting design guideline of allowing the construction by refinement of proved correct programs.

Functional programming offers a higher level of abstraction than most other classical programming paradigms, especially than the imperative paradigm. Furthermore, it is certainly one of the most fruitful means of expression for inventing and studying algorithms, which is of prime importance in the case of quantum computing. A natural way to try and understand precisely how this programming style can be transposed to quantum computing is to study a quantum version of lambda-calculus.

This what is being done by A. Van Tonder at Brown University. His approach puts forward the fact that there is a need for new semantic bases in order to accommodate disturbing peculiarities of the quantum world. A striking example are the consequences of no cloning. In quantum programs, there are quantum variables, i.e.

variables storing quantum states. However, since it is impossible to duplicate the state of a qubit, it is impossible to copy the value of a quantum variable into another quantum variable. This has far reaching consequences, e.g., in lambda-calculus, an impossibility to stay with classical beta-reduction. Van Tonder and J.Y. Girard are suggesting that linear logic may be the way out of this specifically quantum issue.

On the functional side, there is also QPL (a Quantum Programming Language), designed by P. Selinger at the University of Ottawa. QPL is a simple quantum programming language with high-level features such as loops, recursive procedures, and structured data types. The language is functional in nature, statically typed, free of run-time errors, and it has an interesting denotational semantics in terms of complete partial orders of superoperators (superoperators are a generalization of quantum operations).

Process calculi are an abstraction of communicating and cooperating computations which take place during the execution of parallel and distributed programs. They form a natural basis for rigorous and high level expression of several key aspects of quantum information processing: cooperation between quantum and classical parts of a computation, multi-party quantum computation, description and use of teleportation and of its generalization, description and analysis of quantum communication and cryptographic protocols.

CQP (Communicating Quantum Processes) is being designed by R. Nagarayan at the University of Warwick. It combines the communication primitives of the pi-calculus with primitives for measurement and transformation of quantum states. A strong point of CQP is its static type system which classifies channels, distinguishes between quantum and classical data, and controls the use of quantum states: this type system guarantees that each qubit is owned by a unique process within a system.

QPAlg (Quantum Process Algebra) is being designed by M. Lalire and Ph. Jorrand at the University of Grenoble. It does not have yet any elaborate typing system like that of CQP. But, in addition to modelling systems which combine quantum and classical communication and computation, the distinctive features of QPAlg are the explicit representation of qubit initialization through classical to quantum communication, and of measurement through quantum to classical communication, which are systematic and faithful abstractions of physical reality. Similarly, quantum to quantum communication actually models the sending of qubits (not of qubit states) and guarantees that the no cloning law is enforced.

Both CQP and QPAlg have formally defined operational semantics, in the Plotkin's inference rules style, which include a treatment of probabilistic transitions due to the measurement postulate of quantum mechanics.

All these language designs are still at the stage of promising work in progress. The core issues clearly remain at the semantics level, because of the many non-classical properties of the quantum world. No-cloning, entanglement, probabilistic measurement, mixed states (a more abstract view of quantum states, for representing probabilistic distributions over pure states), together with the necessary presence of both worlds, classical and quantum, within a same program, call for further in depth studies toward new bases for adequate semantic frameworks.

Operational semantics (i.e. a formal description of how a quantum + classical program operates) is the easiest part, although probabilities, no-cloning and entanglement already require a form of quantumized treatment. For example, leaving the scope of a quantum variable is not as easy as leaving the scope of a classical variable, since the state of the former may be entangled with the state of more global variables.

Axiomatic semantics (what does a program do? How to reason about it? How to analyse its properties, its behaviour?) is a very tricky part. Defining quantum versions of Hoare's logic or Dijkstra's weakest precondition would indeed provide logical means for reasoning on quantum + classical programs and constitute formal bases for developing such programs. Some attempts toward a dynamic quantum logic, based on the logical study of quantum mechanics initiated in the thirties by Birkhoff and von Neumann have already been made, for example by Brunet and Jorrand, but such approaches rely upon the use of orthomodular logic, which is extremely uneasy to manipulate. Of much relevance here is the recent work of D'Hondt and Panangaden on quantum weakest preconditions, which establishes a semantic universe where programs written in QPL can be interpreted in a very elegant manner.

A long-term goal is the definition of a compositional denotational semantics which would accommodate quantum as well as classical data and operations and provide an answer to the question: what is a quantum + classical program, which mathematical object does it stand for? Working toward this objective has been rather successfully attempted by P. Selinger with QPL. Recent results on categorical semantics for quantum information processing by Abramsky and Coecke, and other different approaches like the work of van Tonder and the interesting manuscript of J. Y. Girard on the relations between quantum computing and linear logic, are also worth considering for further research in those directions.

In fact, there are still a great number of wide open issues in the domain of languages for quantum programming and of their semantics. Two seemingly elementary examples show that there still is a lot to accomplish. First example: classical variables take classical values, quantum variables take quantum states. What would a type system look like for languages allowing both, and which quantum specific properties can be taken care of by such a type system? Second example: it would be very useful to know in advance whether the states of quantum variables will or will not be entangled during execution. Abstract interpretation would the natural approach to answer such a question, but is there an adequate abstraction of quantum states for representing entanglement structures? At the current stage of research in quantum information processing and communication, these and many other similarly interesting questions remain to be solved.

4 Short bibliography

About Quantum Computing

A concise, well written and easy to read (by computer scientists) introduction to quantum computing:

E. G. Rieffel and W. Polak. *An Introduction to Quantum Computing for Non-Physicists*, Los Alamos ArXiv e-print, http://xxx.lanl.gov/abs/quant-ph/9809016, 1998.

A pedagogical, detailed and rather thorough textbook on quantum computing:

M. A. Nielsen and I. L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2000.

A dense and theoretically profound textbook on quantum computing:

A. Y. Kitaev, A. H. Shen and M. N. Vyalyi. *Classical and Quantum Computation*. American Mathematical Society, Graduate Studies in Mathematics, Vol. 47, 2002.

About Quantum Programming

A recent workshop on Quantum Programming Languages:

Second International Workshop on Quantum Programming Languages, held in Turku, Finland, July 12-13, 2004. The full papers are available in the proceedings: http://quasar.mathstat.uottawa.ca/~selinger/qpl2004/proceedings.html, 2004.

Quantum programming languages which have been mentioned in the text:

Imperative quantum programming:

B. Ömer. *Quantum Programming in QCL*. Master's thesis, Institute of Information Systems, Technical University of Vienna, 2000.

P. Zuliani. *Quantum Programming*. PhD thesis, St. Cross College, Oxford University, 2001.

Functional quantum programming:

P. Selinger. *Toward a Quantum Programming Language*. To appear in Mathematical Structures in Computer Science, Cambridge University Press, 2004.

A. Van Tonder. *A lambda Calculus for Quantum Computation*. Los Alamos arXiv e-print, http://xxx.lanl.gov/abs/quant-ph/0307150, 2003.

Parallel and distributed quantum programming:

S. J. Gay and R. Nagarajan. *Communicating Quantum Processes*. In Proc. 2nd International Workshop on Quantum Programming Languages, Turku, Finland, http://quasar.mathstat.uottawa.ca/~selinger/qpl2004/proceedings.html, pp. 91-107, 2004.

M. Lalire and Ph. Jorrand. *A Process Algebraic Approach to Concurrent and Distributed Quantum Computation: Operational Semantics*. Los Alamos arXiv e-print, http://xxx.lanl.gov/abs/quant-ph/0407005, 2004. Also in Proc. 2nd International Workshop on Quantum Programming Languages, Turku, Finland, http://quasar.mathstat.uottawa.ca/~selinger/qpl2004/proceedings.html, pp. 109-126, 2004.

Quantum semantics issues (see also Proc. Int. Workshop on Quantum Programming Languages: http://quasar.mathstat.uottawa.ca/~selinger/qpl2004/proceedings.html):

S. Abramsky and B. Coecke. *Physical Traces: Quantum vs. Classical Information Processing*. In R. Blute and P. Selinger, editors, Proceedings of Category Theory and Computer Science, CTCS'02, ENTCS 69, Elsevier, 2003.

O. Brunet and Ph. Jorrand. *Dynamic Quantum Logic for Quantum Programs*. International J. of Quantum Information, Vol. 2, N. 1, pp. 45-54, 2004.

E. D'Hondt and P. Panangaden. *Quantum Weakest Preconditions*. In Proc. 2nd International Workshop on Quantum Programming Languages, Turku, Finland, http://quasar.mathstat.uottawa.ca/~selinger/qpl2004/proceedings.html, pp. 75-90, 2004.

J. Y. Girard. *Between Logic and Quantic: a Tract*. Manuscript, October 2003.

A. Van Tonder. *Quantum Computation, Categorical Semantics and Linear Logic*. Los Alamos arXiv e-print, http://xxx.lanl.gov/abs/quant-ph/0312174, 2003.

Abstractions for Directing Self-organising Patterns

Daniel Coore

The University of the West Indies, Mona Campus, Jamaica

Abstract. We present an abstraction for pattern formation, called *pattern networks*, which are suitable for constructing complex patterns from simpler ones in the amorphous computing environment. This work builds upon previous efforts that focused on creating suitable system-level abstractions for engineering the emergence of agent-level interactions. Our pattern networks are built up from combinations of these system-level abstractions, and may be combined to form bigger pattern networks. We demonstrate the power of this abstraction by illustrating how a few complex patterns could be generated by a combination of appropriately defined pattern networks. We conclude with a discussion of the challenges involved in parameterising these abstractions, and in defining higher-order versions of them.

1 Introduction

An amorphous computing system is a collection of irregularly placed, locally interacting, identically-programmed, asynchronous computing elements[1]. We assume that these elements (agents) communicate within a fixed radius, which is large relative to the size of an element, but small relative to the diameter of the system. We also assume that most agents do not initially have any information to distinguish themselves from other agents; this includes information such as position, identity and connectivity. The challenge of amorphous computing is to systematically produce a program that when executed by each agent (in parallel) produces some pre-specified system-wide behaviour.

The model is motivated by advances in microfabrication and in cellular engineering [7, 3] – technologies that will enable us to build systems with more parts than we currently can. In fact, we envision these technologies allowing for systems containing myriads (think hundreds of thousands) of elements to be built cheaply. The technology for precisely controlling such systems is still in its infancy, yet we can observe natural systems with similar complexity operating with apparently high efficiency. Compelling examples include: cells in an embryo specialising to give rise to the form and function of the various parts of an organism; ants, bees and other social insects cooperating in various ways to find food for the colony; and birds flocking to reduce drag during flight.

The amorphous computing model is not explicit about the assumed capabilities of a single agent – only that it has limited memory to work with because of its size. It has become commonplace to assume that each agent, operating in isolation, can:

– maintain and update state (read and write to memory)
– access a timer (a piece of state that changes uniformly with time, but not necessarily at the same rate as happens on other agents)
– access a source of random bits
– perform simple arithmetic and logic operations.

So far, one of the most important results of Amorphous Computing is that despite the constraints of the computing model, in which so little positional information is available, it is still possible to engineer the emergence of certain types of globally defined patterns [6, 9, 4, 8, 2]. In each of these works, the principal mechanism used to control emergence was to define a system-level language that could describe an interesting class of patterns, but that could also be systematically transformed to agent-level computations.

1.1 A Unifying Pattern Description Language

Recently, we proposed a unifying language[5] that is capable of describing the same patterns as those produced by the programming languages defined in [6, 9, 4, 8]. In this language, the system is regarded as a collection of points. Each point has a *neighbourhood* which is the set of points that lie within some fixed distance of the original point. Points can be named and some operations may yield points as results, which may be used in any context in which a point is expected. We can execute blocks of commands at any point. Computations that are specified to occur at multiple points are performed in parallel, but generally computations specified at a single point are carried out sequentially (although threaded computations are permitted at a point).

All computations are either entirely internal to an agent or may be derived from local communication between neighbouring agents. All primitive operations, whether they can be computed by an agent in isolation or only through neighbourhoods, have a mechanism for indicating to the originating point when the computation is considered completed. This means that any operations performed at a point may be sequenced together, even if the operation involves computations at many points.

2 Pattern Networks

The recently proposed pattern description language[5] is not rich in abstractions. It provides primitives that abstract over the agent-level interactions, but it does not provide any abstractions over the patterns that may be described. For example, it is possible to describe four lines that form a square, but it is not possible to name that pattern of four lines and invoke it whenever we need a square. Instead we must describe the four constituent lines, each time a square is desired.

We propose a plausible abstraction for patterns, called a *pattern network*, that is compatible with this language. This abstraction is similar to the network abstractions for GPL described in[6]. The implementation of this abstraction in our pattern description language is in progress, so any sample outputs illustrating the idea have been taken from the GPL version of the abstraction.

Pattern networks describe patterns in terms of a set of given points in the domain (the *input* points). Special points that arise from the generation of these patterns (the *output* points) may be named and used as inputs to other pattern networks. The `network` construct defines a pattern network. Its syntax is:

```
network ⟨name⟩ [⟨inputs⟩] [⟨outputs⟩] {
      ⟨pattern-defn⟩
}
```

2.1 An Example

As an illustration of how this abstraction can be used, let us reuse the code, presented in [5], that draws a line segment between two given points. We present below, the original code surrounded by a box to highlight the minimal syntax of the pattern network abstraction.

```
network segment[A, B] [] {
   Diffusable B-stuff
   PointSet ABLine
   at B: diffuse (B-stuff, linear-unbounded)
   at A:do SearchForB {
        addTo(ABLine)
        nbrConcs := filter(queryNbhd(B-stuff),
                           lt(B-stuff))
        at select(nbrConcs, min):
        do SearchForB
   }
}
```

In this example, the network is called `segment` and has two input points (`A`, `B`) and zero output points. Whenever the `segment` network is invoked, a logical substance named `B-stuff` is diffused from the point supplied as `B`. Simultaneously from `A`, a process is started that locally seeks the source of the `B-stuff` diffusion, and propagates from point to point until it is found. This process labels each visited point with the `ABLine` label to identify the line segment that would have formed from `A` to `B` after the process is completed. In this example, both the logical substance `B-stuff` and the set label `ABLine` are local to the network, so they are actually renamed on each invocation of the network so that the actual labels exchanged at the agent level are actually unique to the invocation of the network. This avoids unwanted interference between separate invocations of the same network. Now, for example, to define a triangle between three points, we might define the following network:

```
network triangle [A, B, C] [] {
   segment[A,B]
   segment[B,C]
   segment[C,A]
}
```

2.2 Combining Patterns

Pattern networks can be defined in terms of combinations of smaller pattern networks. We have already seen an example of this in the `triangle` network defined above, however that was simply the simultaneous invocation of three instances of the `segment` network. We can go further by using the output of one network to feed into another. A special operation has to be introduced for associating output points of one network with inputs of another. We introduced the `cascade` operation to conveniently implement the simple chaining of networks together. Its syntax is:

cascade(⟨ *points* ⟩, ⟨ *net*$_1$ ⟩, ..., ⟨ *net*$_n$ ⟩, [*points*])

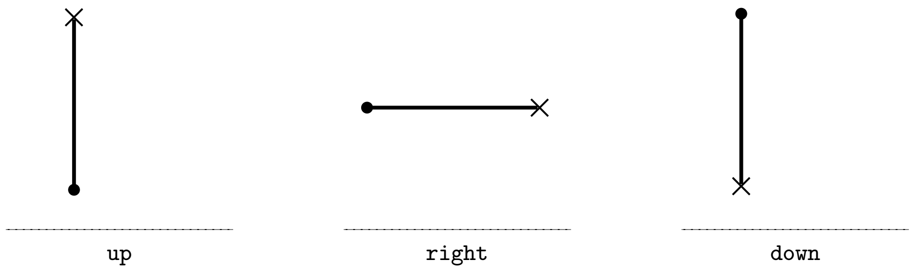
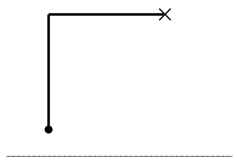


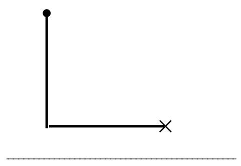
Fig. 1. Three networks, each having one input and one output have been defined, and named *up*, *right* and *down*. Each one draws a short line in the indicated direction relative to a reference line (produced by some other pattern). In each case, a dot indicates the location of the input point and an 'x' indicates the output point

To illustrate the operation of `cascade`, let us assume that we already have three pattern networks, each with one input and one output, named `up`, `right` and `down`. Each one draws a short line relative to some reference line (constructed by some other pattern) in the indicated direction. The termination point of each line is the output point of its network. These networks are illustrated in Figure 1. We could now define a new network, called `rising` with one input and one output that is the result of cascading `up` with `right`. A similar network, called `falling`, could be composed from cascading `down` with `right`. To get one period of a square wave, we could `cascade rising` and `falling` together. The effect of the `cascade` operation in these definitions is illustrated in Figure 2.

```
network rising [A][B] {
      cascade([A], up, right, [B])
}
```



```
network falling [A][B] {
      cascade([A], down, right, [B])
}
```



```
network sq-period [A][B] {
      cascade([A], rising, falling, [B])
}
```

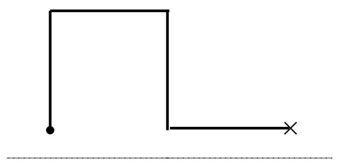


Fig. 2. The composition of networks with the `cascade` operation. Pattern networks can be composed and abstracted to build compound patterns that would otherwise be tedious to construct.

We could now further build upon this network by constructing a square wave of several periods, by cascading many instances of `sq-period` together. It should be clear that the `cascade` operation has provided considerable expressive power by enabling the composition of a large pattern from sub-patterns.

When there are multiple inputs and outputs, the cascade operation associates outputs with inputs according to the order in which they appear. In some instances, the user may want the association between outputs of one network and inputs of another to be some permutation other than the identity permutation. This re-ordering of points can be achieved with the use of a binding construct that permits intermediate points to be named and used in any context in which a point expression may legally appear. The `let-points` construct facilitates this; its syntax is defined as:

```
let-points [⟨ id1 ⟩, ..., ⟨ idn ⟩]
              {⟨ point-defns ⟩}
in {⟨ pattern-defn ⟩}
```

The `point-defns` expression may be any expression that is capable of yielding points as outputs. These points are bound to the names `id`$_1$`,` `id`$_2$ for use in evaluating the `pattern-defn` expression, which itself is also any expression that may have legally appeared in the body of a pattern network. The `let-points` construct allows outputs to be permuted when cascaded with other networks.

For example, suppose that `net1` and `net2` are two pattern networks each with two inputs and two outputs. Now, suppose we want to cascade the outputs of net1 to the inputs of net2 but with the orderings exchanged (i.e. first output to second input and vice-versa). In addition, we shall make the connected structure a new pattern network. The following network definition uses the let-points construct to accomplish this.

```
network crossed [A, B][C, D] {
   let-points[out1, out2]
             {cascade([A, B], net1, [out1, out2])}
   in {cascade([out2, out1], net2, [C, D])}
}
```

The let-points command not only provides a generic means for supporting arbitrary permutations of output points, but it also provides a means for connecting the outputs of two pattern networks to the inputs of a single network. The following example shows how the outputs from the right and up networks (previously mentioned) to the inputs of net1. It creates a pattern network that has two inputs and two outputs. Each input point acts as the input to the single-input networks, their outputs are connected to the inputs of net1 and its outputs are the outputs of the overall network.

```
network two-to-one [A, B][C, D] {
   let-points[out1, out2]
             {
              cascade([A], right, [out1])
              cascade([B], up, [out2])
             }
   in {cascade([out1, out2], net1, [C, D])}
   }
```

3 Examples

Pattern networks provide a convenient high-level abstraction for describing patterns in a system-level language. In this section, we illustrate their versatility by showing how they were used in GPL to describe self-organising text and CMOS circuit layouts.

3.1 Text

As an example, let us use pattern networks to define patterns of self-organising text. The idea is that each letter of the alphabet is defined as a pattern network. Each letter is defined relative to a line, which will be generated along with the characters themselves. Each network has two input points and two output points. One input point defines the starting point for the letter, and the other the starting point for the segment of the reference line, upon which the letter sits. The letters are drawn from left to right, so the outputs represent the locations that a letter placed to the right of the current one would use as its inputs. In this way, a word can be defined as a cascade of pattern networks each of which produces a single letter. Furthermore, each word is also now a network with exactly two inputs and two outputs that can be cascaded with other words and letters to form bigger networks. Figure 3 illustrates the result of using this technique in GPL with three letters, each separately defined as a network and then cascaded together.

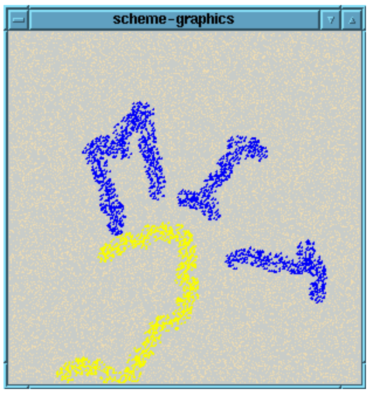


Fig. 3. The result of cascading the pattern networks for the letters ‘M’, ‘I’ and ‘T’ together. Observe that the reference line is subject to the variability in the distribution of the agents. A badly drawn reference line can often cause the desired letters to fail to form in a recognizable way

3.2 CMOS Circuits

Another fruitful family of patterns to which pattern networks could be applied with great effect is that of circuit diagrams. Indeed, input and output points for a circuit’s pattern network often directly correspond to the input and output terminals of the circuit. For example, the pattern representing the CMOS layout of an AND gate can be described as the cascade of the layouts of a NAND gate, a via (a connector across layers in the CMOS process) and an inverter. In GPL, this network is expressed as follows:

```
(define-network (and+rails
                 (vdd-in vss-in a b)
                 (vdd-out vss-out output))
  (cascade (vdd-in vss-in a b)
           nand+rails
           via+rails
           inverter+rails
           (vdd-out vss-out output)))
```

Figure 4 illustrates how the various constituent networks of the AND circuit are combined.

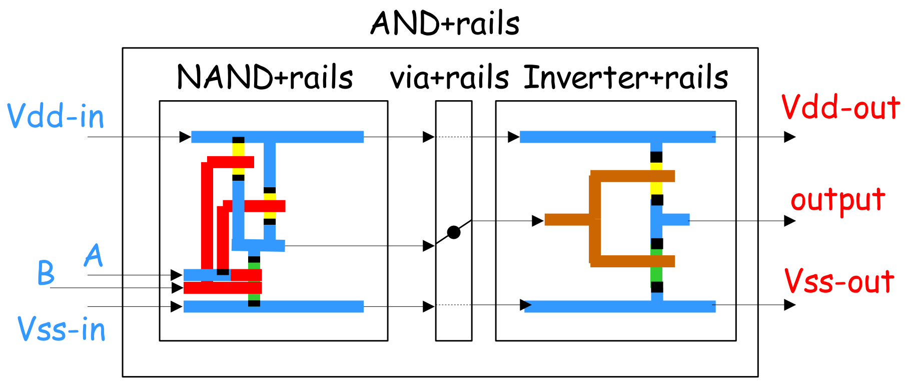


Fig. 4. The *NAND+rails* network has 4 inputs and 3 outputs, and both of *via+rails* and *Inverter+rails* have 3 inputs and 3 outputs. The arrows depict the connections between the outputs and inputs of the three networks. Each bordering rectangle depicts a network. Observe that the result of the cascade of the three networks is itself another network with 4 inputs (the number of inputs of the first network in the cascade) and 3 outputs (the number of outputs of the last network in the cascade)

4 Challenges for Extensions

Since the input points involved in the definition of a pattern network, are not necessarily local to each other, the invocation of a network's computation is actually the parallel invocation of all computations specified at the input points. In particular, the output points that result from this parallel execution may also be non-local to each other. This poses some problems for attempting to create parameterised or higher-order abstractions of networks. For example, if pattern networks could accept parameters, their values would have to be available to all points participating in the invocation of these networks. This could probably be accomplished by passing around an environment specific to the pattern network, but represents a significant departure from the current macro-style implementation of pattern networks. Supporting higher-order pattern networks requires allowing the invocation of networks passed as parameters or the creation of networks on the fly. Both of these requirements require signficant changes to the current implementation of pattern networks in GPL though, so the idea of higher-level pattern networks has not yet been explored fully.

5 Conclusions

Pattern networks provide a powerful means of expressing patterns that can be self-organised. They can capture simple patterns that are directly expressed in terms of local interactions, and they can be combined to form more complex pattern networks. In addition, the expression of these combinations are quite natural, yet they remain transformable to collections of local interactions that, under the appropriate conditions, cause the desired patterns to emerge. Pattern networks are currently not generalisable to higher order patterns nor can they even accept parameters to allow more general specifications. However, these limitations are currently self-imposed as a simplification exercise, and we expect some of them, at least, will be overcome in the near future.

References

1. Abelson, H., D. Allen, D. Coore, C. Hanson, G. Homsy, T. Knight, R. Nagpal, E. Rauch, G. Sussman, and R. Weiss. Amorphous Computing. *Communications of the ACM*, **43**(5), May 2000.
2. Beal, Jacob, "Persistent Nodes for Reliable Memory in Geographically Local Networks", *AI Memo 2003-011*, Massachusetts Institute of Technology, Artificial Intelligence Laboratory. Apr. 2003.
3. Benenson, Yaakov., Rivka Adar,Tamar Paz-Elizur, Zvi Livneh, and Ehud Shapiro, "DNA molecule provides a computing machine with both data and fuel." in *Proceedings National Academy of Science, USA* **100**, 2191-2196. (2003).
4. Clement, Lauren, and Radhika Nagpal, " Self-Assembly and Self-Repairing Topologies", in *Workshop on Adaptability in Multi-Agent Systems, RoboCup Australian Open*, Jan. 2003.
5. Coore, Daniel, "Towards a Universal Language for Amorphous Computing", in *International Conference on Complex Systems (ICCS2004)*, May 2004.
6. Coore, Daniel, *Botanical Computing: A Developmental Approach to Generating Interconnect Topologies on an Amorphous Computer*. PhD thesis, MIT, Dept. of Electrical Engineering and Computer Science, Feb. 1999.
7. Knight, Tom, Gerald J. Sussman, "Cellular Gate Technologies", in *First International Conference on Unconventional Models of Computation (UMC98)*, 1998.
8. Kondacs, Attila, "Biologically-Inspired Self-Assembly of Two-Dimensional Shapes Using Global-to-Local Compilation", in *International Joint Conference on Artificial Intelligence (IJCAI)*, Aug. 2003.
9. Nagpal, Radhika, *Programmable Self-Assembly: Constructing Global Shape using Biologically-inspired Local Interactions and Origami Mathematics*. PhD thesis, MIT, Dept. of Electrical Engineering and Computer Science, (June 2001).

Abstractions for code reuse in ECOLI

Dean D. Holness

The University of the West Indies, Mona Campus, Jamaica

Abstract. Many programming paradigms have ways of permitting reuse of code.For example in the object oriented paradigm, code reuse can be achieved through the use of inheritance, methods and instances. This paper describes three abstractions that facilitate code reuse in an amorphous programming language. These abstractions have significantly reduced the size and complexity of amorphous programs and facilitate the design of efficient and robust algorithms. We also present an implementation of these abstractions in the amorphous programming language, ECOLI. The ECOLI programming language has many similarities to the object-oriented paradigm, so do the abstractions put forward by this paper. Several examples are also presented to show the effectiveness of these abstractions. We argue that they are necessary for any amorphous programming language to realistically solve non-trivial problems.

1 Introduction

The re-use of code is one of the many ways in which software developers reduce the amount of time needed to implement a given solution. Many programming paradigms have recognized this and have introduced language constructs, to facilitate the design and implementation of reusable software components. This concept is also very applicable to the Amorphous paradigm [?] since in many cases the code produced in solving problems will usually have fragments in common. It is from these commonalities between programs, that we have developed a series of abstractions in order to better incorporate the code reuse process in Amorphous programming languages. We have further implemented these abstractions as constructs in our own Amorphous programming language ECOLI (Extensible Calculus of Local Interactions) and have seen where it has had significant benefits in the design and implementation of our algorithms. The central problem with code reuse in an Amorphous setting is capturing the repeating portion and expressing it. This paper puts forward three abstractions for code reuse, *Instances*, *Inheritance and Remote procedure calls.* These abstractions were inspired by analogies from the Object Oriented paradigm, and for good reason. The Amorphous programming paradigm has various similarities with Object Oriented design. In object oriented design, objects are exclusive entity with communication only occurring through messages passed between them. So is it too with the Amorphous paradigm, each agent in the system is an exclusive entities with communication only occurring through some local interaction.

2 ECOLI (Extensible Calculus of Local Interactions)

The ECOLI programming language was first described by Daniel Coore in his PhD Thesis [?]. ECOLI is meant to be a language for programming the computational element of an Amorphous Computer. ECOLI assumes that the particles are irregularly placed and the communication between the particles is asynchronous and occurs concurrently with their computation. It also assumes that communication between agents occur via messages and the delivery time of each message is proportional to its length and at most one message can be received in one message delay time interval. This imposes a serial nature on the agent's ability to process messages. In keeping with the Amorphous model of computational [?], every agent executes the same program.

2.1 Structure of ECOLI

An ECOLI program consists of components called *Behaviors, Dictionaries and Handlers.* In essence an ECOLI program is a collection of possible responses to messages. These responses are what we call *Handlers.* Handlers can be grouped to form *Dictionaries*, so that each dictionary defines the set of messages to which the agent can respond while it is active. The ability of an agent to differentiate its behavior over time comes from the fact that the current dictionary can be programmatically changed. The meaning of a message is therefore, dependent on the agent's current dictionary and could then differ from agent to agent. A useful analogy would be to a finite state machine where the destinations of the transitions are determined during execution. In this sense a behaviour defines a the state machine. A dictionary would represent a state, each transition from the state is represented by a handler and the input to the state machine is a message. Messages sent from one agent's behavior, will be received by the same behavior on the neighboring agents. A behavior defines a specific high level task in an Amorphous environment, whether it is to draw a line between two points or to find the distances from a point. Here is an example of a behavior.

ECOLI Behaviour code fragment

```
(define-behavior
    find-dist
  ((current-dist 10000))
(define-dict default
   (DIST
    (n)
    (guard (< n current-dist))
    (set current-dist n)
    (trap current-dist)
    (send DIST (+ n 1))
    )
```

```
    (ACTIVATE
    ()
    (set current-dist 0)
    (send DIST 1)
    )
   )
 )
```

In the preceding example, the behavior *find-dist* is designed to find the distances of agents from a given point. It contains one dictionary, *default* and two handlers, *DIST* and *ACTIVATE*. The program starts by assigning a specified agent the role of being the center, from which all other measurements will be taken. The starting processor is sent an ACTIVATE message, which in turn sends a DIST message with *1* as its argument. The neighbors of the starting processor will receive the DIST message and invoke their own DIST handler, which will add one to the argument and resend the message to their neighbors. The special keyword, *guard* is used to determine if the agent should respond to the message. In this case, the guarding condition is that the agent's value for the variable *current-dist* is greater than the argument of the message. At the end of the program, all agents in reach will have a value for the current-dist variable which represents the minimum number of communication hops between it and the starting agent.

3 ECOLI Abstractions

This section puts forward our abstraction for code reuse in ECOLI, along with examples of actual programs that have benefited from these abstractions.

3.1 Dictionary Inheritance

In ECOLI, each dictionary defines a handler for each message to which the agent can potentially respond while the dictionary is active. If the agent wants to respond to one message differently but keep all the handlers for the other messages the same, it must define a new dictionary. Since a dictionary must define handlers, for all possible messages, the programmer would then be forced to define, not only the changed handler but also the unchanged ones. Therefore, the code for the two dictionaries would be exactly the same, except the changed handler. We have defined an abstraction that eliminates the need for this handler duplication. It allows the programmer to re-use the previous dictionary's handler definitions and redefine only the ones that are different. Dictionary inheritance is similar to inheritance in the object oriented paradigm, where a class inherits all the behaviors of its parent class and redefines a behavior only if it changes. An ECOLI programmer can now specify that a dictionary /em extends a previous dictionary, and with that, the new dictionary will automatically gain access to all the parent dictionary's handler definitions. This significantly reduces the size of ECOLI programs, since it has been empirically found that dictionaries share a number of handlers in common. The following examples illustrate the use of dictionary inheritance. The first code segment describes an implementation without the use of inheritance and the second, with dictionary inheritance.

Inheritance - code fragment 1

```
(define-behavior
    original-dict
  ()
  (define-dict original
   (HANDLER1
    (src)
    (trap src)
    (send HANDLER1 (+ src 10))
    )
   (HANDLER2
    (src)
    (trap src)
    (send HANDLER2 (* src 20))
    )
   (HANDLER3
    (src)
    (trap src)
    (send HANDLER3 (/ src 50))
    )
   (ACTIVATE
   ()
   (set current-dist 0)
   (trap current-dist)
   (send DIST my-id 1)
   )
  )
```

```
(define-dict derived
   (HANDLER1
    (src)
    (trap src)
    (send HANDLER1 (+ src 10))
    )
   (HANDLER2
    (src)
    (trap src)
    (send HANDLER2 (* src 20))
    )
   (HANDLER3
    (src)
    (trap src)
    (send HANDLER3 (+ src (/ src 50)))
    )
   (ACTIVATE
   ()
   (set current-dist 0)
   (trap current-dist)
   (send DIST my-id 1)
   )
)
```

Inheritance - code fragment 2

```
(define-behavior
  extenddict
  ()
  (define-dict original-dict
   (HANDLER1
    (src)
    (trap src)
    (send HANDLER1 (+ src 10))
    )
   (HANDLER2
    (src)
    (trap src)
    (send HANDLER2 (* src 20))
    )
   (HANDLER3
    (src)
    (trap src)
    (send HANDLER3 (/ src 50))
    )
   (ACTIVATE
   ()
   (set current-dist 0)
   (trap current-dist)
   (send DIST my-id 1)
   )
  )
```

```
(define-dict derived (extend original-dict
  (HANDLER3
   (src)
   (trap src)
   (send HANDLER3 (+ src (/ src 50)))))
  )
```

3.2 Remote Procedure Calls

Behaviors define specific tasks in an ECOLI program. It would be useful if one behavior could be called from another behavior as a subroutine in an aid to solve a smaller problem on which the calling behavior can build. There are numerous things to consider when trying to achieve this. First, there is no easy way to suspend the execution of a behavior to facilitate the running of the subroutine. Second, the mechanisms for detecting when a behavior is completed and informing the calling behavior are tedious and unreliable. We have introduced a Remote Procedure Call (RPC) like construct, that allows a behavior to call another behavior as a subroutine, allow the subroutine to perform its task and return to a specified handler in the calling behavior. It is similar to remote procedure calls in that the calling behavior essentially invokes the desire behavior on its neighbors. A typical example is a solution for drawing a square given a behavior that defines a line-segment. ideally we would like to call the line-drawing behavior as a subroutine to the square-drawing behavior. The following code fragment illustrates the use of the *call and return* construct.

Remote Procedure Calls - code fragment

```
(define-behavior
    run-prog
  (define-dict
    (ACTIVATE
     ()
     (if b-point?
     (begin
       (set B-material true)
       (call RETURNHERE find-dist ACTIVATE)))
    )
    (RETURNHERE
     ()
     (send RETURNER)
     )
  )
```

The code illustrates how a call to a handler in the behavior can be implemented as a subroutine. In the above case, the *run-prog* behavior calls the *find-dist* behavior as a subroutine. The called behavior will execute and when finished, will pass control back to the *run-prog* behavior, more specifically to the *RETURN-HERE* handler.

3.3 Instances of Behaviors

Each behavior usually represents a solution to a specific task in ECOLI, and is analogous to an object in the object oriented paradigm. The behavior's state is represented by its state variables and its current dictionary. Recall that all agents in the amorphous environment execute the same program. Therefore, if one agent starts a behavior to solve a particular problem for its own use, and another agent starts the same behavior for a completely different problem. The behaviors, though logically trying to achieve different versions of the same task, still have the same name and since messages sent from an agent are delivered to the handler of the behavior with the same name on its neighbors, there is no way to stop the two executions of the behavior from interfering. Observe the following: An ECOLI programmer has defined a behavior that models the secretion process of a particular chemical A. The behavior contains the concentration of Chemical A as its state variable, and the process of secreting is modeled by the behavior's dictionaries and handlers. The programmer now decides to implement a process that models the secretion of a new chemical B. Even though the process for secreting Chemical A and B is identical, to stop the behaviors from interfering, the programmer would have to implement an exact duplicate of the behavior to secrete Chemical A with the only difference being the behavior's name. We have borrowed from the Object Oriented paradigm again in a bid to stem this behavior duplication issue. In object oriented design, the implementation of a solution is done only once, in a class, and all further uses of that implementation simply creates an instance of the class. We can do the same thing here. If we abstract away the elements of a behavior that makes it specific, and focus on the commonalities between them, then we can then define instances of that behavior to accomplish specific tasks. Since these instances are unique then they will be able to execute concurrently on the amorphous media without the risk of interference from each other. In the example above, the programmer would simple make an instance of the secrete behavior for chemical A, and another instance for chemical B. From the agents perspective, these instances are treated as behaviors with separate names. Hence messages sent from these instances will be delivered to the appropriate instances in the neighboring agent. Here is an example of the use of instances, and how the concept was implemented in ECOLI before its introduction. The first code fragment demonstrates why instances are important. It illustrates how the programmer would need to explicitly create a different behavior for every specific execution of a secretion task. Whereas in the second code fragment, the programmer simply defines the secrete behavior once, and makes as many instances as necessary from it.

Instances - code fragment 1

```
(define-behavior
  secrete-A-pheromone
    ...
    (UPDATE
     (val)
     (guard (and (> val value)))
     (set B-pheromone instance)
     (set value val)
     (send UPDATE (- value 1))
     )
    (POLL
     ()
     (send SENDVALUES my-id)
     )
    )
  )
```

```
(define-behavior
  secrete-B-pheromone
  (define-dict default
     ....
     (UPDATE
     (val)
     (guard (and (> val value)))
     (set B-pheromone instance)
     (set value val)
     (send UPDATE (- value 1))
     )
    (POLL
     ()
     (send SENDVALUES my-id)
     )
    )
  )
```

```
(define-behavior
    run-prog
  (define-dict
    (ACTIVATE
     ()
     (if b-point?
     (begin
       (set B-material true)
       (call AFTERA secrete-B-pheromone ACTIVATE)))
     (if a-point?
     (begin
       (set A-material true)
       (call AFTERB secrete-A-pheromone ACTIVATE)))
     )
    )
  )
```

Instances - code fragment 2

```
(define-behavior
  secrete
    ...
    (UPDATE
     (val)
     (guard (and (> val value)))
     (set B-pheromone instance)
     (set value val)
     (send UPDATE (- value 1))
     )
    (POLL
     ()
     (send SENDVALUES my-id)
     )
    )
  )
```

```
(define-behavior
    run-prog
  (define-dict
    (ACTIVATE
     ()
     (if b-point?
     (begin
       (set B-pheromone (new secrete))
       (call AFTERB B-pheromone ACTIVATE)))
     (if a-point?
     (begin
       (set A-pheromone (new secrete))
       (call AFTERA A-pheromone ACTIVATE)))
     )
    )
  )
```

References

1. Coore, Daniel, *Botanical Computing: A Developmental Approach to Generating Interconnect Topologies on an Amorphous Computer*. PhD thesis, MIT, Dept. of Electrical Engineering and Computer Science, Feb 1999.
2. Abelson, H., D. Allen. D. Coore, C. Homsy, T. Knight, R. Nagpal, E. Rauch, G. Sussman, and R. Weiss. Amorphous Computing. Communications of the ACM, May 2000.
3. Campbell,Roy H., Gary M. Johnston, Peter W. Madany, Vincent F. Russo. *Principles of Object-Oriented Operating System Design* (1991)
4. Shaw Mary, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, Gregory Zelesnik. *Abstractions for Software Architecture and Tools to Support Them* (1995)
5. Neighbors, James M., *Software Construction Using Components* (1980)

Programming an Amorphous Computational Medium

Jacob Beal

Massachusetts Institute of Technology, Cambridge MA 02139, USA

Amorphous computing considers the problem of controlling millions of spatially distributed unreliable devices which communicate only to nearby neighbors. To program such a system, we need a high-level description language for desired global behaviors, and a system to compile such descriptions into locally executing code which then robustly creates and maintains the desired global behavior. I survey the existing computational primitives, give desiderata for a language describing computation on an amorphous medium, and sketch how the existing primitives might be combined to produce an language that computes on an amorphous computer as though it were a space-filling computational medium.

1 The Amorphous Computing Scenario

The amorphous computing engineering domain presents a set of challenging requirements and prohibitions to the system designer, forcing confrontation of issues of robustness, distribution, and scalability. These constraints derive much of their inspiration from biological systems engaged in morphogenesis and regrowth, which must be accomplished by coordinating extremely large numbers of unreliable devices (cells).

The first and foremost requirement is **scalability**: the number of devices may be large, anywhere from thousands to millions or even billions. Biological systems, in fact, may comprise trillions of cells. Practically, this means that an algorithm is only reasonable if its per-device asymptotic complexity (e.g. space or bandwidth per device) are polynomial in $\log n$ (where n is the number of devices) — and any bound significantly greater than $O(lgn)$ should be treated with considerable suspicion. Further, unlike many ad-hoc networking domains, amorphous computing generally assumes **cheap energy, local processing, and storage** — in other words, as long as they do not have a high per-device asymptotic complexity, minimizing them is not of particular interest.

The network graph is determined by the **spatial distribution** of the devices in some Euclidean space, which collaborate via **local communication**. Devices are generally **immobile** unless the space in which they are embedded is moved (e.g. cutting and pasting "smart paper"). A unit disc model is often used to create the network, in which a bidirectional link exists between two devices if and only if they are less than distance r apart. Local communication implies that the network is expected to have a high diameter, and, assuming a packet-based communication model, this means that the time for information to propagate through the network depends primarily on the number of hops it needs to travel.

Local communication also means that communication complexity is best measured by **maximum communication density** — the maximum bandwidth required by a device — rather than by number of messages.

Due to the number of devices potentially involved, the system must not depend on much care and feeding of the individual devices. In general, it is assumed that every device is **identically programmed**, but that there can be a small amount of **differentiation via initial conditions**. Once the system is running, however, there is **no per-device administration** allowed.

There are strict limitations on the assumed network infrastructure. The system executes with **partial synchrony** — each device may be assumed to have a clock which ticks regularly, but the clocks may show different times, run at (boundedly) different rates, and have different phases. In addition, the system may not assume complex services not presently extended to the amorphous domain — this is applied particularly to mean **no global naming, routing, or coordinate service** may be assumed.

Finally, in a large, spatially distributed network of devices, failures are not isolated events. Due to the sheer number of devices, point failures are best measured by an expected **rate of device failure**. This suggests also that methods of analysis like half-life analysis[9] will be more useful than standard f-failure analysis. In addition, because the network is spatially embedded, outside events may cause **failure of arbitrary spatial regions** — larger region failures are assumed to occur less frequently. Generally stopping failures are have been used, in which the failing device or link simple ceases operating. Finally, maintenance requires recovery or replacement of failed devices: in either case, the effect is that **new devices join** the network either individually or as a region.

2 Existing Amorphous Computing Primitives

Several existing amorphous computing algorithms will serve as useful primitives for constructing a language. Each algorithm summarized here has been implemented as a module of code and demonstrated in simulation.

2.1 Shared Neighborhood Data

This simple module allows devices to communicate by means of a shared-memory region. Each device maintains a table of key-value pairs which it wishes to share. Periodically each device transmits its table to its neighbors, informing them that it is still a neighbor and refreshing their view of its shared memory. Conversely, a neighbor is removed from the table if more than a certain time has elapsed since its last refresh. The module can then be queried for the set of neighbors, and the values its neighbors most recently held for any key in its table.

Maintaining shared neighborhood data takes requires storage and communication density proportional to the amount of data being shared.

2.2 Regions

The region module maintains labels for contiguous sets of devices. A Region is defined by a name and a membership test. When seeded in one or more devices, a Region spreads via shared neighborhood data to all adjoining nodes that satisfy the membership test. When a Region is deallocated, a garbage collection mechanism spreads the deallocation throughout the participating devices, attempting to ensure that the defunct Region is removed totally.

Note that failures or evolving system state may separate a Region into disconnected components. While these are still logically the same Region, and may rejoin into a single connected component in the future, information may not pass between disconnected components. As a result, the state of disconnected components of a Region may evolve separately, and in particular garbage collection is only guaranteed to be effective in a connected component of a Region.

Regions are organized into a tree, with every device belonging to the root region. In order for a device to be a member of a region, it must also be a member of that region's parent in the tree. This implicit compounding of membership tests allows regions to exhibit stack-like behavior which will be useful for establishing execution scope in a high-level language.

Maintaining Regions requires storage and communication density proportional to the number of Regions being maintained, due to the maintenance of shared neighborhood data. Garbage collecting a Region requires time proportional to the diameter of the Region.

2.3 Broadcast/Convergecast

The broadcast/convergecast module does a best-effort transmission of an acknowledged message to every device within a Region. For a broadcast, the acknowledgement is just an indicator allowing the broadcast operation to terminate; for a convergecast the acknowledgement merges the result of a query. The broadcast builds a multi-tree as information propagates outward, which resolves into a tree as the acknowledgements return, then is garbage-collected after the acknowledgement is delivered.

The broadcast module uses a port abstraction, so many processes can use the broadcast module without interference. Each running broadcast requires constant storage and communication density within the Region in which it executes, and takes time proportional to the diameter of the region.

Broadcast/convergecast is of limited use for a high-level language because it executes communication between a single device and a large group of devices. Gossip and Consensus/Reduction are more suitable for many purposes, but may be more expensive.

2.4 Gossip

The gossip communication module[10, 5] propagates information throughout a Region via shared neighborhood data. Unlike a broadcast, gossip is all-to-all communication: each item of gossip has a merge function combines local state with neighbor information to produce a merged whole. When an item of gossip is garbage-collected, the deallocation propagates slowly to prevent regrowth into areas which have already been garbage-collected.

Gossip requires storage and communication density proportional to the number and size of gossip items being maintained in each Region of which a device is a member, due to the maintenance of shared neighborhood data. Garbage collecting an item of gossip takes time proportional to the diameter of the region.

2.5 Consensus (Reduction)

Devices participating in a consensus process must all choose the same value if any of them choose a value, and the chosen value must be held by at least one of the participants. Reduction is a generalization of consensus in which the chosen value is a function of all values held by the participants (e.g. sum or average).

The Paxos consensus algorithm[8] has been demonstrated in an amorphous computing context,[5] but scales very badly. A gossip-based algorithm currently under development promises much better results: it appears that running a reduction process on a Region will require storage and communication density logarithmic in the diameter of the Region and time log-linear in the diameter of the Region.

2.6 Gradient

A gradient[6, 7] counts upwards from its source or sources — a set of devices which declare themselves to have count value zero — giving an approximate spherical distance measure useful for establishing regions. The count is one more than the minimum value in the neighborhood, so the gradient converges to the shortest distance over time. The gradient runs within a Region, and may count only a bounded number of hops. When the supporting sources disappear, the gradient is garbage-collected; as in the case of gossip items, the garbage collection propagates slowly to prevent regrowth. A gradient may also carry version information, allowing its source to change rapidly by increasing the version number.

Maintaining a gradient requires a constant amount of storage and communication density for every device in range of the gradient, and garbage collecting a gradient takes time linear in the minimum of its bound or the diameter of the Region in which it is running.

2.7 Persistent Node

A Persistent Node[2] is a distributed object based around a versioned gradient. The gradient flows outward from the center, identifying all devices within r hops (the node's **core**) as members of the Persistent Node, while a heuristic calculation flows inward from all devices within $2r$ hops (the node's **reflector**) to determine which direction the center should be moving. The gradient is bounded to kr hops (the node's **umbra**), so every device in this region is aware of the existence and location of the Persistent Node. If any device in the node core survives a failure, the node will rebuild itself, although if the failure separates the core into disconnected components, the node may be cloned. If the umbras of two clones ever come in contact, however, they will resolve back into a single node via the destruction of one clone.

The node is identified as a Region whose parent is the Region in which its gradient runs. In addition, a node is a read/write object supporting conditionally atomic transactions.

Maintaining a persistent node requires storage and communication density linear in the size of the data stored by the node. The node moves and repairs itself in time linear in the diameter of the node.

2.8 Hierarchical Partitioning

A Region can be partitioned through the use of Persistent Nodes:[3] nodes of a characteristic radius are generated and allowed to drift, repelling one another, until every device in the region is near some Persistent Node, and devices choose a node to associate with, with some hysteresis. A set of partitions with exponentially increasing diameter can be wired together to form a hierarchical partition of the Region which is highly resilient to failures.

Maintaining a hierarchical partition takes storage and communication logarithmic in the diameter of the Region being partitioned, and creating the partition takes time log-linear in the diameter.

2.9 Failure Circumscription

Failure circumscription[4] identifies a connected Region containing a group of failures, given a pre-existing hierarchical partitioning. Devices bordering the failure climb the hierarchy until they find a clique of neighbors which are all either still alive or provably dead: such a clique circumscribes the failure and the lowest such clique is near optimal for convex failures. By identifying a circumscribing Region, exception handling might be limited to the area local to the failure.

Finding the Region which circumscribes a failure requires space and communication density logarithmic in the diameter of the failure or optimal circumscription, whichever is greater, and requires time linear in the diameter.

3 Amorphous Medium Language

I want to be able to program an amorphous computing system as though it is a space filled with a continuous medium of computational material: the actual executing program should produce an approximation of this behavior on a set of discrete points. This means that there should be no explicit statements about individual devices or communication between them. Instead, the language should describe behavior in terms of spatial regions.

The program should be able to be specified without knowledge of the particular amorphous medium on which it will be run. Moreover, given the shape of the medium should be able to change as the program is executing, through failure and addition of devices, and the running program adjust to its new environment without undue disruption.

Finally, since failures and additions may disconnect and reconnect the medium, a program which is separated into two different executions must be able to reintegrate when the components of the medium rejoin.

The modules outlined in the previous section appear to be a sufficient set of primitives from which such a high-level Amorphous Medium Language may be derived. The tree of Regions can serve as stack frames for the execution, with branches implementing parallelism — both distributed processing (non-overlapping regions) and multithreading (a device which is a member of multiple branches). Variables within a stack frame may be implemented by gossip or by a consensus/reduction process. Defining a subspace to execute a function call can be done by a Region membership test, or by specifying geometric areas with Persistent Nodes and gradients.

I propose that the actual computation should be specified not as imperative instructions, but as invariants to be repaired. A process contains a set of constraint specifications for the invariants to be maintained. If the constraint is satisfied, then no computation need take place. Each constraint specification should also include a repair rule which is executed to move towards satisfying the constraint at each step. Thus execution of the system is better described not as correct operation, but rather convergence from an incorrect state toward a correct goal. This model allows for transparent adaptation to change and recovery from error: failures are not exceptions, but merely setbacks that place the system farther from convergence.

3.1 Example: Partitioning a Region

Figure 3.1 shows tentative pseudocode for an AML implementation of the partition subprocess used in building a hierarchical partition. There are four constraints defining the invariants for such a partition: every device should be near a partitioning persistent node, the persistent nodes defining the partition should not overlap, a device chooses (with hysteresis) the closest persistent node as its partition, and each component has a set of neighbors equal to its center. For each constraint, there is a remedy to be executed when the constraint is violated. For example, if there is no partition persistent node to join, the device flips a coin to see if it should start one. If, on the other hand, the device is in the core of two or more nodes, it will kill off all but the highest precedence one. Information about the neighbors is spread outward from the center of the component's associated persistent node by a gossip function, and the heuristic dictates that partition persistent nodes will avoid one another. Finally, each component is, itself, a Region, which may have other processes executed within it.

```
;;;; Partition Process:
(process
 partition
 (r) ;; one parameter: the characteristic radius for partitioning
 (component neighbors) ;; two local vbls: the component and its neighbors

 ;; persistent nodes should move to avoid each other
 (defun partition-fn ()
   (node-function
    (avoid (get-nodes :type (partition r) :reflector))))

 ;; neighbor information spreads by gossip
 (gossip neighbors :region component :merge #'newer)

  ;; every devices should be covered by a persistent node
 (constraint
  (get-nodes :type (partition r) :radii 2.5)
  (when (< (random) (/ 1 (* (square (* 3 r)) (* 1.5 r))))
    (new-node :type (partition r) :motion partition-fn)))

 ;; if two persistent nodes are too close together, one should suicide
 (constraint
  (<= (length (get-nodes :type (partition r) :core) 1))
  (let ((survivor (max (mapcar #'node-precedence
                               (get-nodes :type (partition r) :core)))))
   (map #'kill-node
     (remove survivor (get-nodes :type (partition r) :core)
             :key #'node-precedence))))

 ;; a device should belong to the component of the closest
 ;; persistent node (w. hysteresis)
 (constraint
  (and component
       (< (- (radius (get-node component)) 0.5)
          (min (map #'radius
                    (get-nodes :type (partition r) :radii 2.5)))))
  (let ((best (reduce #'best-radius
                      (get-nodes :type (partition r) :radii 2.5))))
    (setf component best)))

 ;; neighbors are the persistent nodes w. umbras touching the center
 (constraint
  (or (not (centerp component))
      (equal nil (set-difference (cons component neighbors)
                                 (get-nodes :type (partition r)))))
  (setf neighbors (remove component (get-nodes :type (partition r))))))
```

Fig. 1. Tentative pseudocode for partition a region into subregions with characteristic radius r, a subroutine used for hierarchical partitioning.

References

1. H. Abelson, D. Allen, D. Coore, C. Hanson, G. Homsy, T. Knight, R. Nagpal, E. Rauch, G. Sussman and R. Weiss. Amorphous computing. AI Memo 1665, MIT, 1999.
2. J. Beal. Persistent nodes for reliable memory in geographically local networks. Tech Report AIM-2003-11, MIT, 2003.
3. J. Beal. A robust amorphous hierarchy from persistent nodes. In *CSN*, 2003.
4. J. Beal. Near-optimal distributed failure circumscription. In *PDCS*, 2003.
5. J. Beal, S. Gilbert. RamboNodes for the metropolitan ad hoc network. Tech Report AIM-2003-027, MIT, 2003.
6. Daniel Coore. Establishing a Coordinate System on an Amorphous Computer. MIT Student Workshop on High Performance Computing, 1998.
7. Daniel Coore, Radhika Nagpal and Ron Weiss. Paradigms for structure in an amorphous computer. MIT AI Memo 1614.
8. L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.
9. D. Liben-Nowell, H. Balakrishnan, D. Karger. Analysis of the evolution of peer-to-peer systems. In *PODC*, 2002.
10. N. Lynch. *Distributed Algorithms.* Morgan Kaufman, 1996.

Computations in Space and Space in Computations

Olivier Michel, Jean-Louis Giavitto, Julien Cohen, Antoine Spicher
LaMI[*]– CNRS – Université d'Évry – Genopole

(*draft paper*)

The Analytical Engine weaves algebraic patterns just as the Jacquard loom weaves flowers and leaves.
Ada Lovelace

1 Goals and Motivations

The emergence of terms like *natural computing*, *mimetic computing*, *parallel problem solving from nature*, *bio-inspired computing*, *neurocomputing*, *evolutionary computing*, etc., shows the never ending interest of the computer scientists for the use of "natural phenomena" as "problem solving devices" or more generally, as a fruitful source of inspiration to develop new programming paradigms. It is the latter topic which interests us here. The idea of *numerical experiment* can be reversed and, instead of using computers to simulate a fragment of the real world, the idea is to use (a digital simulation of) the real world to compute. In this perspective, the processes that take place in the real world are the objects of a new calculus:

description of the world's laws	=	program
state of the world	=	data of the program
parameters of the description	=	inputs of the program
simulation	=	the computation

This approach can be summarized by the following slogan: "programming *in* the language of nature" and was present since the very beginning 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 approach offers many advantages from the *teaching*, *heuristic* and *technical* points of view: it is easier to explain concepts referring to real world processes that are actual examples; the analogy with the nature acts as a powerful source of inspirations; and the studies of natural phenomena by the various scientific disciplines (physics, biology, chemistry...) have elaborated a large body of concepts and tools that can be used to study computations (some concrete examples of this cross fertilization relying on the concept of dynamical system are given in references [6, 5, 32, 11]).

There is a *possible fallacy* in this perspective: the description of the nature is not unique and diverse concurent approaches have been developed to account for the same objects. Therefore, there is not a unique "language of nature" prescribing a unique and definitive programming paradigm. *However*, there is a common concern shared by the various descriptions of nature provided by the scientific disciplines: *natural phenomena take place in time and space.*

In this paper, we propose the use of spatial notions as structuring relationships *in* a programming language. Considering space in a computation is hardly new: the use of spatial (and temporal) notions is at the basis of computational complexity *of* a program; spatial and temporal relationships are also used in the implementation of parallel languages (if two computations occur at the same time, then the two computations must be located at two different places, which is the basic constraint that drives the scheduling and the data distribution problems in parallel programming); the methods for building domains in denotational semantics have also clearly topological roots, but they involve the *topology of the set of*

[*]LaMI umr 8042 CNRS – Université d'Évry, Tour Évry-2, 523 place des terrasses de l'agora, 91000 Évry, France. Emails: `[michel, giavitto, jcohen, aspicher]@lami.univ-evry.fr`
This paper is a short version of a LaMI technical report.

values, not the *topology of a value*. In summary, spatial notions have been so far mainly used to describe the running of a program and not as *means to develop new programs*.

We want to stress this last point of view: we are not concerned by the organization of the resources used by a program run. What we want is to develop a spatial point of view on the entities built by the programmer when he designs his programs. From this perspective, a program must be seen as a space where computation occurs and a computation can be structured by spatial relationships. We hope to provide some evidences in the rest of this paper that the concept of space can be as fertile as mathematical logic for the development of programming languages. More specifically, we advocate that the concepts and tools developed for the algebraic construction and characterization of shapes[1] provide interesting teaching, heuristic and technical alternatives to develop new data structures and new control structures for programming.

The rest of this paper is organized as follows. Section 2 and section 3 provide an informal discussion to convince the reader of the interest of introducing a topological point of view in programming. This approach is illustrated through the experimental programming language MGS used as a vehicle to investigate and validate the topological approach.

Section 2 introduces the idea of seeing a data structure as a space where the computation and the values move. Section 3 follows the spatial metaphor and presents control structures as path specifications. The previous ideas underlie MGS. Section 4 sketches this language. The presentation is restricted to the notions needed to follow the examples in the next section. Section 5 gives some examples and introduces the $(\text{DS})^2$ class of dynamical systems which exhibit a dynamical structure. Such kind of systems are hard to model and simulate because the state space must be computed jointly with the running state of the system. To conclude in section 6 we indicate some of the related work and we mention briefly some perspectives on the use of spatial notions.

2 Data Structures as Spaces[2]

The relative accessibility from one element to another is a key point considered in a data structure definition:

- In a simply linked list, the elements are accessed linearly (the second after the first, the third after the second, etc.).
- In a circular buffer, or in a double-linked list, the computation goes from one element to the following *or* to the previous one.
- From a node in a tree, we can access the sons.
- The neighbors of a vertex V in a graph are visited after V when traveling through the graph.
- In a record, the various fields are locally related and this localization can be named by an identifier.
- Neighborhood relationships between array elements are left implicit in the array data-structure. Implementing neighborhood on arrays relies on an index algebra: index computations are used to code the access to a neighbor. The standard example of index algebra is integer tuples with linear mappings $\lambda x.x \pm 1$ along each dimension (called "Von Neumann" or "Moore" neighborhoods).

This accessibility relation defines a logical neighborhood. The concept of logical neighborhood in a data structure is not only an abstraction perceived by the programmer and vanishing at the execution, but it does have an actual meaning for the computation. Very often the computation indeed complies with the logical neighborhood of the data elements and it is folk's knowledge that most of the algorithms are structured either following the structure of the input data or the structure of the output data. Let us give some examples.

[1] G. Gaston-Granger in [22] considers three avenues in the formalization of the concept of space: *shape* (the algebraic construction and the transformation of space and spatial configurations), *texture* (the continuum) and *measure* (the process of counting and coordinatization [37]). In this work, we rely on elementary concepts developed in the field of combinatorial algebraic topology for the construction of spaces [23].

[2] The ideas exposed in this section are developed in [18, 13].

The recursive definition of the `fold` function on lists propagates an action to be performed along the traversal of a list. More generally, recursive computations on data structures respect so often the logical neighborhood, that standard high-order functions (e.g. *primitive recursion*) can be automatically defined from the data structure organization (think about catamorphisms and other polytypic functions on inductive types [27, 24]).

The list of examples can be continued to convince ourselves that a notion of logical neighborhood is fundamental in the definition of a data structure. So to define a data organization, we adopt a *topological* point of view: *a data structure can be seen as a space*, the set of positions between which *the computation moves. Each position possibly holds a value*[3]. The set of positions is called the *container* and the values labeling the positions constitute the *content*.

This topological approach is constructive: one can define a data type by the set of moves allowed in the data structure. An example is given by the notion of "Group Based Fields" or GBF in short [20, 15]. In a uniform data structure, i.e. in a data structure where any elementary move can be used against any position, the set of moves possesses the structure of a mathematical group $\mathcal{G}$. The neighborhood relationship of the container corresponds to the Cayley graph of $\mathcal{G}$. In this paper, we will use only two very simple groups $\mathcal{G}$ corresponding to the moves `|north>` and `|east>` allowed in the usual two-dimensional grid and to the moves allowed in the hexagonal lattice figured at the right of Fig. 3.

3 Control Structures as Paths

In the previous section, we suggested looking at data structure as spaces in which computation moves. Then, when the computation proceeds, a path in the data structure is traversed. This path is driven by the control structures of the program. So, a control structure can be seen as a path specification in the space of a data structure. We elaborate on this idea into two directions: concurrent processes and multi-agent systems.

3.1 Homotopy of a Program Run

Consider two sequential processes `A` and `B` that share a semaphore s. The current state of the parallel execution `P = A || B` can be figured as a point in the plane $A \times B$ where A (resp. B) is the sequence of instructions of `A` (resp. `B`). Thus, the running of `P` corresponds to a path in the plane $A \times B$. However, there are two constraints on paths that represent the execution of `P`. Such a path must be "increasing" because we suppose that at least one of the two subprocesses `A` or `B` must progress. The second constraint is that the two subprocesses cannot be simultaneously in the region protected by the semaphore s. This constraint has a clear geometrical interpretation: the increasing paths must avoid an "obstruction region", see Fig. 1. Such representation is known at least from the 1970's as "progress graph" [7] and is used to study the possible deadlocks of a set of concurrent processes.

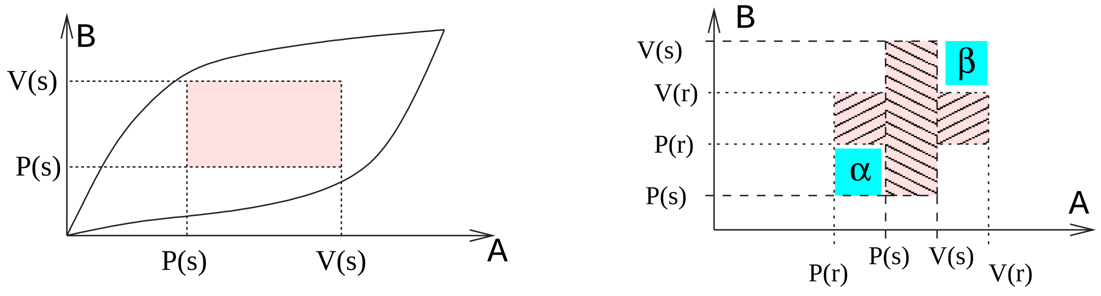


Figure 1: *Left:* The possible path taken by the process `A || B` is constrained by the obstruction resulting of a semaphore shared between the processes `A` and `B`. *Right:* The sharing of two semaphores between two processes may lead to deadlock (corresponding to the domain α) or to the existence of a "garden of Eden" (the domain β that cannot be accessed outside from β and it can only be leaved.)

[3] *A point in space is a placeholder awaiting for an argument*, L. Wittgenstein, (Tractatus Logico Philosophicus, 2.0131).

Homotopy (the continuous deformation of a path) can be adapted to take into account the constraint of increasing paths and provides effective tools to detect deadlocks or to classify the behavior of a parallel program (for instance in the previous example, there are two classes of paths corresponding to executions where the process `A` or `B` enters the semaphore first). Refer to [21] for an introduction to this domain.

3.2 The Topological Structure of Interactions[4]

In a multi-agent system (or an object based or an actor system), the control structures are less explicit and the emphasis is put on the local interaction between two (sometimes more) agents. In this section, we want to show that the interactions between the elements of a system exhibit a natural topology.

The starting point is the decomposition of a system into subsystems defined by the requirement that the elements into the subsystems interact together and are truly independent from all other subsystems parallel evolution.

In this view, the decomposition of a system S into subsystems $S_1, S_2, \ldots, S_n$ is *functional*: state $s_i(t+1)$ of the subsystem S_i depends solely of the previous state $s_i(t)$. However, the decomposition of S into the S_i can depend on the time steps. So we write $S_1^t, S_2^t, \ldots, S_{n_t}^t$ for the decomposition of the system S at time t and we have: $s_i(t+1) = h_i^t(s_i(t))$ where the h_i^t are the "local" evolution functions of the S_i^t. The "global" state $s(t)$ of the system S can be recovered from the "local" states of the subsystems: there is a function φ^t such that $s(t) = \varphi^t(s_1(t), \ldots, s_{n_t}(t))$ which induces a relation between the "global" evolution function h and the local evolution functions: $s(t+1) = h(s(t)) = \varphi^t(h_1^t(s_1(t)), \ldots, h_{n_t}^t(s_{n_t}(t)))$.

The successive decomposition $S_1^t, S_2^t, \ldots, S_{n_t}^t$ can be used to capture the *elementary parts* and the *interaction structure* between these elementary parts of S. Cf. Figure 2. Two subsystems S' and S'' of S interact if there are some S_j^t such that $S', S'' \in S_j^t$. Two subsystems S' and S'' are *separable* if there are some S_j^t such that $S' \in S_j^t$ and $S'' \notin S_j^t$ or vice-versa. This leads to consider the set $\mathcal{S}$, called the *interaction structure* of S, defined by the smaller set closed by intersection that contains the S_j^t.

Set $\mathcal{S}$ has a *topological structure*: $\mathcal{S}$ corresponds to an *abstract simplicial complex*. An abstract simplicial complex [23] is a collection $\mathcal{S}$ of finite non-empty set such that if A is an element of $\mathcal{S}$, so is every nonempty subset of A. The element A of $\mathcal{S}$ is called a *simplex* of $\mathcal{S}$; its *dimension* is one less that the number of its elements. The dimension of $\mathcal{S}$ is the largest dimension of one of its simplices. Each nonempty subset of A is called a *face* and the *vertex set* $V(\mathcal{S})$, defined by the union of the one point elements of $\mathcal{S}$, corresponds to the *elementary functional parts* of the system S. The abstract simplicial complex notion generalizes the idea of *graph*: a simplex of dimension 1 is an edge that links two vertices, a simplex f of dimension 2 can be thought of as a surface whose boundaries are the simplices of dimension 1 included in f, etc.

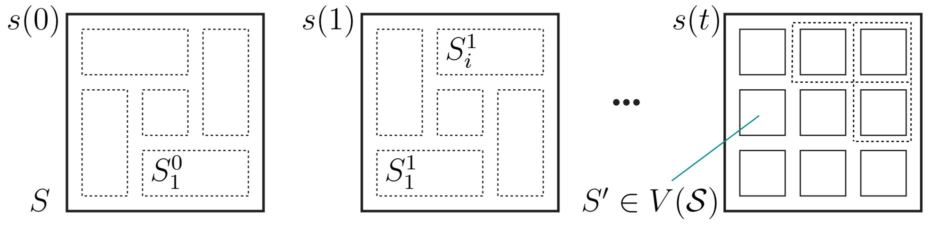


Figure 2: The interaction structure of a system S resulting from the subsystems of elements in interaction at a given time step.

4 MGS Principles

The two previous sections give several examples to convince the reader that a topological approach of the data and control structures of a program present some interesting perspectives for language design: a data structure can be defined as a space (and there are many ways to build spaces) and a control structure is a path specification (and there are many ways to specify a path).

[4]This section is adapted from [34].

Such a topological approach is at the core of the MGS project. Starting from the analysis of the interaction structure in the previous section, our idea is to define directly the set $\mathcal{S}$ with its topological structure and to specify the evolution function h by specifying the set S_i^t and the functions h_i^t:

- the interaction structure $\mathcal{S}$ is defined as a new kind of data structures called *topological collections*;
- a set of functions h_i^t together with the specification of the S_i^t for a given t are called a *transformation*.

We will show that this abstract approach enables an homogeneous and uniform handling of several computational models including cellular automata (CA), lattice gas automata, abstract chemistry, Lindenmayer systems, Paun systems and several other abstract reduction systems.

These ideas are validated by the development of a language also called MGS. This language embeds a complete, strict, impure, dynamically or statically typed functional language.

4.1 Topological Collections

The distinctive feature of the MGS language is its handling of entities structured by *abstract topologies* using *transformations* [19]. A set of entities organized by an abstract topology is called a *topological collection*. Here, topological means that each collection type defines a neighborhood relation inducing a notion of *subcollection*. A subcollection S' of a collection S is a subset of connected elements of S and inheriting its organization from S. Beware that by "neighborhood relation" we simply mean a relationship that specify if two elements are neighbors. From this relation, a cellular complex can be built and the classical "neighborhood structure" in terms of open and closed sets can be recovered [33].

Collection Types. Different predefined and user-defined collection types are available in MGS, including sets, bags (or multisets), sequences, Cayley graphs of Abelian groups (which include several unbounded, circular and twisted grids), Delaunay triangulations, arbitrary graphs, quasi-manifolds [34] and some other arbitrary topologies specified by the programmer.

Building Topological Collections. For any collection type `T`, the corresponding empty collection is written `():T`. The join of two collections C_1 and C_2 (written by a comma: C_1,C_2) is the main operation on collections. The comma operator is overloaded in MGS and can be used to build any collection (the type of the arguments disambiguates the collection built). So, the expression `1, 1+2, 2+1, ():set` builds the set with the two elements `1` and `3`, while the expression `1, 1+2, 2+1, ():bag` computes a bag (a set that allows multiple occurrences of the same value) with the three elements `1`, `3` and `3`. A set or a bag is provided with the following topology: in a set or a bag, any two elements are neighbors. To spare the notations, the empty sequence can be omitted in the definition of a sequence: `1, 2, 3` is equivalent to `1, 2, 3,():seq`.

4.2 Transformations

The MGS experimental programming language implements the idea of transformations of topological collections into the framework of a functional language: collections are just new kinds of values and transformations are functions acting on collections and defined by a specific syntax using rules. Transformations (like functions) are first-class values and can be passed as arguments or returned as the result of an application.

The *global transformation* of a topological collection s consists in the *parallel application* of a set of *local transformations*. A local transformation is specified by a rule r that specifies the replacement of a subcollection by another one. The application of a rewriting rule $\sigma \Rightarrow f(\sigma, ...)$ to a collection s:

1. selects a subcollection s_i of s whose elements match the *pattern* σ,
2. computes a new collection s'_i as a function f of s_i and its neighbors,
3. and specifies the insertion of s'_i in place of s_i into s.

One should pay attention to the fact that, due to the parallel application strategy of rules, *all distinct instances s_i of the subcollections matched by the σ pattern are "simultaneously replaced" by the $f(s_i)$.*

Path Pattern. A pattern σ in the left hand side of a rule specifies a subcollection where an interaction occurs. A subcollection of interacting elements can have an arbitrary shape, making it very difficult to specify. Thus, it is more convenient (and not so restrictive) to enumerate sequentially the elements of the subcollection. Such enumeration will be called a *path*.

A path pattern *Pat* is a sequence or a repetition *Rep* of *basic filters*. A basic filter *BF* matches one element. The following (fragment of the) grammar of path patterns reflects this decomposition:

$$\mathit{Pat} ::= \mathit{Rep} \mid \mathit{Rep}\ \texttt{,}\ \mathit{Pat} \qquad \mathit{Rep} ::= \mathit{BF} \mid \mathit{BF}\texttt{/}\mathit{exp} \qquad \mathit{BF} ::= \texttt{cte} \mid \mathrm{id} \mid \texttt{<undef>}$$

where `cte` is a literal value, id ranges over the pattern variables and *exp* is a boolean expression. The following explanations give a systematic interpretation for these patterns:

literal: a literal value `cte` matches an element with the same value.

empty element the symbol `<undef>` matches an element whose position does not have an associated value.

variable: a pattern variable a matches exactly one element with a well defined value. The variable a can then occur elsewhere in the rest of pattern or in the r.h.s. of the rule and denotes the value of the matched element.

neighbor: b, p is a pattern that matches a path which begins by an element matched by b and continues by a path matched by p, the first element of p being a neighbor of b.

guard: p/exp matches a path matched by p when the boolean expression *exp* evaluates to `true`.

Elements matched by basic filters in a rule are distinct. So a matched path is without self-intersection. The identifier of a pattern variable can be used only once as a basic filter. That is, the path pattern x,x is forbidden. However, this pattern can be rewritten for instance as: x,y / y=x.

Right Hand Side of a Rule. The right hand side of a rule specifies a collection that replaces the subcollection matched by the pattern in the left hand side. There is an alternative point of view: because the pattern defines a sequence of elements, the right hand side may be an expression that evaluates to a sequence of elements. Then, the substitution is done element-wise: element i in the matched path is replaced by the element i in the r.h.s. This point of view enables a very concise writing of the rules.

5 Examples

5.1 The modeling of Dynamical Systems

In this section, we show through one example the ability of MGS to concisely and easily express the state of a dynamical system and its evolution function. More examples can be found on the MGS web page and include: cellular automata-like examples (game of life, snowflake formation, lattice gas automata...), various resolutions of partial differential equations (like the diffusion-reaction *à la* Turing), Lindenmayer systems (e.g. the modeling of the heterocysts differentiation during Anabaena growth), the modeling of a spatially distributed signaling pathway, the flocking of birds, the modeling of a tumor growth, the growth of a meristeme, the simulation of colonies of ants foraging for food, etc.

The example given below is an example of a discrete "classical dynamical system". We term it "classical" because it exhibits a *static structure*: the state of the system is statically described and does not change with the time. This situation is simple and arises often in elementary physics. For example, a falling stone is statically described by a position and a velocity and this set of variables does not change (even if the value of the position and the value of the velocity change in the course of time). However, in some systems, it is not only the values of state variables, but also the *set* of state variables *and/or* the evolution function, that changes over time. We call these systems *dynamical systems with a dynamic structure* following [16], or (DS)^2 in short. As pointed out by [14], many biological systems are of this kind. The rationale and the use of MGS in the simulation of (DS)^2 is presented in [13, 14].

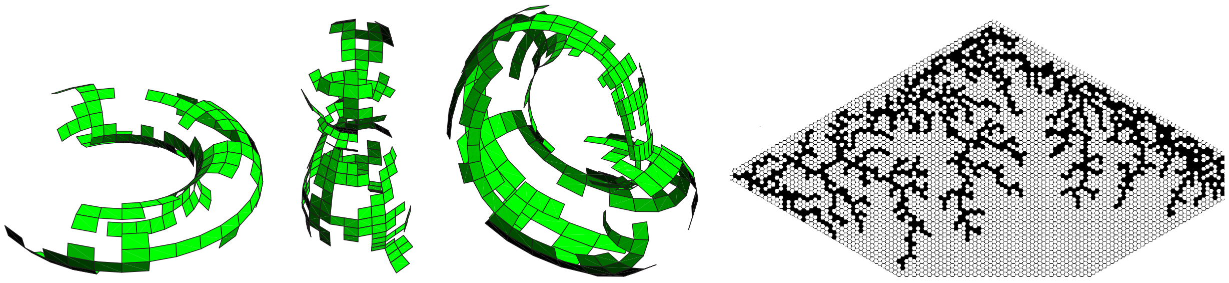


Figure 3: From left to right: the final state of a DLA process on a torus, a chess pawn, a Klein's bottle and an hexagonal meshes. The chess pawn is homeomorphic to a sphere and the Klein's bottle does not admit a concretization in Euclidean space. These two topological collections are values of the *quasi-manifold* type. Such collection are build using *G-map*, a data-structure widely used in geometric modeling [25]. The torus and the hexagonal mesh are GBFs.

Diffusion Limited Aggreation (DLA). DLA, is a fractal growth model studied by T.A. Witten and L.M. Sander, in the eighties. The principle of the model is simple: a set of particles diffuses randomly on a given spatial domain. Initially one particle, the seed, is fixed. When a mobile particle collides a fixed one, they stick together and stay fixed. For the sake of simplicity, we suppose that they stick together forever and that there is no aggregate formation between two mobile particles. This process leads to a simple CA with an asynchronous update function or a lattice gas automata with a slightly more elaborate rule set. This section shows that the MGS approach enables the specification of a simple generic transformation that can act on arbitrary complex topologies.

The transformation describing the DLA behavior is really simple. We use two symbolic values `‘free` and `‘fixed` to represent respectively a mobile and a fixed particle. There are two rules in the transformation:

1. the first rule specifies that if a diffusing particle is the neighbor of a fixed seed, then it becomes fixed (at the current position);
2. the second one specifies the random diffusion process: if a mobile particle is neighbor of an empty place (position), then it may leave its current position to occupy the empty neighbor (and its current position is made empty).

Note that the order of the rules is important because the first has priority over the second one. Thus, we have :

```
trans dla = {
    ‘free, ‘fixed   =>  ‘fixed, ‘fixed
    ‘free, <undef>  =>  <undef>, ‘free
}
```

This transformation is polytypic and can be applied to any kind of collection, see Fig. 3 for a few results.

5.2 Programming in the Small: Algorithmic Examples

The previous section advocates the adequation of the MGS programming style to model and simulate various dynamical systems. However, it appears that the MGS programming style is also well fitted for the implementation of algorithmic tasks. In this section, we show some examples that support this assertion. More examples can be found on the MGS web page and include: the analysis of the Needham-Schroeder public-key protocol [28], the Eratosthene's sieve, the normalization of boolean formulas, the computation of various algorithms on graphs like the computation of the shortest distance between two nodes or the maximal flow, etc.

5.2.1 Gamma and the Chemical Computing Metaphor

In MGS, the topology of a multiset is the topology of a complete connected graph: each element is the neighbor of any other element. With this topology, transformations can be used to easily emulate a Gamma transformations [2, 3]. The Gamma transformation on the left is simply translated into the MGS transformation on the right:

```
M = do                                                    trans M = {
    rp $x_1, \dots, x_n$                                       $x_1, \dots, x_n$
    if $P(x_1, \dots, x_n)$                         ⟹         / $P(x_1, \dots, x_n)$
    by $f_1(x_1, \dots, x_n), \dots, f_m(x_1, \dots, x_n)$     => $f_1(x_1, \dots, x_n), \dots, f_m(x_1, \dots, x_n)$ }
```

and the application `M(b)` of a Gamma transformation `M` to a multiset `b` is replaced in MGS by the computation of the fixpoint iteration `M[iter=‘fixpoint](b)`. The optional parameter `iter` is a system parameter that allows the programmer to choose amongst several predefined application strategies: f `[iter=‘fixpoint]` (x_0) computes $x_1 = f(x_0), x_2 = f(x_1), ..., x_n = f(x_{n-1})$ and returns x_n such that $x_n = x_{n-1}$.

As a consequence, the concise and elegant programming style of Gamma is enabled in MGS: refer to the Gamma literature for numerous examples of algorithms, from knapsack to the maximal convex hull of a set of points, through the computation of prime numbers. See also the numerous applications of multiset rewriting developped in the projects Elan [36] and Maude [35].

One can see MGS as "Gamma with more structure". However, one can note that the topology of a multiset is "universal" in the following sense: it embeds any other neighborhood relationship. So, it is always possible to code (at the price of explicit coding the topological relation into some value inspected at run-time) any specific topology on top of the multiset topology. We interpret the development of "structured Gamma" [10] in this perspective.

5.2.2 Two Sorting Algorithms

A kind of bubble-sort is straightforward in MGS; it is sufficient to specify the exchange of two non-ordered adjacent elements in a sequence, see Fig. 4. The corresponding transformation is defined as:

```
trans BubbleSort = {   x,y / x>y  ⇒  y,x   }
```

The transformation *BubbleSort* must be iterated until a fixpoint is reached. This is not a real a bubble sort algorithm because swapping of elements happen at arbitrary places; hence an out-of-order element does not necessarily bubble to the top in the characteristic way.

Bead sort is a new sorting algorithm [1]. The idea is to represent positive integers by a set of beads, like those used in an abacus. Beads are attached to vertical rods and appear to be suspended in the air just before sliding down (a number is read horizontally, as a row). After their falls, the rows of numbers have been rearranged such as the smaller numbers appears on top of greater numbers, see Fig. 4. The corresponding one-line MGS program is given by the transformation:

```
trans BeadSort = {   'empty |north> 'bead  ⇒  'bead, 'empty   }
```

This transformation is applied on the usual grid. The constant `'empty` is used to give a value to an empty place and the constant `'bead` is used to represent an occupied cell. The l.h.s. of the only rule of the transformation *BeadSort* selects the paths of length two, composed by an occupied cell at north of an empty cell. Such a path is replaced by a path computed in the r.h.s. of the rule. The r.h.s. in this example computes a path of length two with the occupied and the empty cell swapped.

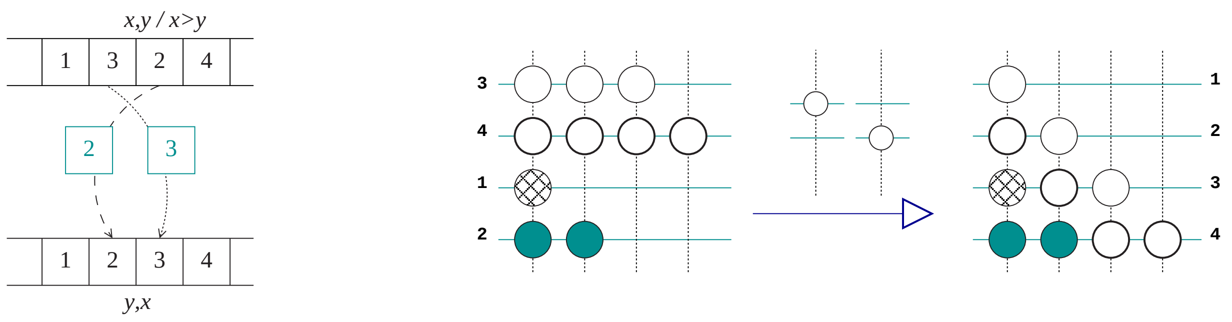


Figure 4: *Left:* Bubble sort. *Right:* Bead sort [1].

5.2.3 Hamiltonian Path.

A graph is a MGS topological collection. It is very easy to list all the Hamiltonian paths in a graph using the transformation:

```
trans H = {
    x* / size(x) = size(self) / Print(x) / false => assert(false)
}
```

This transformation uses an iterated pattern x* that matches a path (a sequence of elements neighbor two by two). The keyword `self` refers to the collection on which the transformation is applied, that is, the all graph. The size of a graph returns the number of its vertices. So, if the length of the path x is the same as the number of vertices in the graph, then the path x is an Hamiltonian path because matched paths are simple (no repetition of an element). The second guard prints the Hamiltonian path as a side effect and returns its argument which is not a false value. Then the third guard is checked and returns false, thus, the r.h.s. of the rule is never triggered. The matching strategy ensures a maximal rule application. In other words, if a rule is not triggered, then there is no instance of a possible path that fulfills the pattern. This property implies that the previous rule must be checked on all possible Hamiltonian paths and H(`g`) prints all the Hamiltonian path in `g` before returning `g` unchanged.

6 Current Status and Related Work

The topological approach we have sketched here is part of a long term research effort [20] developed for instance in [12] where the focus is on the substructure, or in [15] where a general tool for uniform neighborhood definition is developed. Within this long term research project, MGS is an experimental language used to investigate the idea of associating computations to paths through rules. The application of such rules can be seen as a kind of rewriting process on a collection of objects organized by a topological relationship (the neighborhood). A privileged application domain for MGS is the modeling and simulation of dynamical systems that exhibit a dynamic structure.

Multiset transformation is reminiscent of multiset-rewriting (or rewriting of terms modulo AC). This is the main computational device of Gamma [2], a language based on a chemical metaphor; the data are considered as a multiset M of molecules and the computation is a succession of chemical reactions according to a particular rule. The CHemical Abstract Machine (CHAM) extends these ideas with a focus on the expression of semantic of non deterministic processes [4]. The CHAM introduces a mechanism to isolate some parts of the chemical solution. This idea has been seriously taken into account in the notion of P systems. P systems [29] are a recent distributed parallel computing model based on the notion of a membrane structure. A membrane structure is a nesting of cells represented, e.g, by a Venn diagram without intersection and with a unique superset: the skin. Objects are placed in the regions defined by the membranes and evolve following various transformations: an object can evolve into another object, can pass trough a membrane or dissolve its enclosing membrane. As for Gamma, the computation is finished when no object can further evolve. By using nested multisets, MGS is able to emulate more or less the notion of P systems. In addition, patterns like the iteration + go beyond what is possible to specify in the l.h.s. of a Gamma rule.

Lindenmayer systems [26] have long been used in the modeling of (DS)^2 (especially in the modeling of plant growing). They loosely correspond to transformations on sequences or string rewriting (they also correspond to tree rewriting, because some standard features make particularly simple to code arbitrary trees, Cf. the work of P. Prusinkiewicz [30]). Obviously, L systems are dedicated to the handling of linear and tree-like structures.

There are strong links between GBF and cellular automata (CA), especially considering the work of Z. Róka which has studied CA on Cayley graphs [31]. However, our own work focuses on the construction of Cayley graphs as the shape of a data structure and we develop an operator algebra and rewriting notions on this new data type. This is not in the line of Z. Róka which focuses on synchronization problems and establishes complexity results in the framework of CA.

A unifying theoretical framework can be developed [17, 19], based on the notion of *chain complex* developed in algebraic combinatorial topology. However, we do not claim that we have achieved a useful theoretical framework encompassing the previous paradigm. We advocate that few topological notions and a single syntax can be consistently used to allow the merging of these formalisms *for programming* purposes.

The current MGS interpreter is freely available at the MGS home page: `http://mgs.lami.univ-evry.fr`. A compiler is under development where a static type discipline can be enforced [8, 9]) to enforce a static type discipline [8, 9]. There are two versions of the type inference systems for MGS: the first one is a classical extension of the Hindley-Milner type inference system that handles homogeneous collections. The second one is a soft type system able to handle heterogeneous collection (e.g. a sequence containing both integers and booleans is heterogeneous).

Acknowledgments

The authors would like to thanks Franck Delaplace at LaMI, Frédéric Gruau at University of Paris-Sud, Florent Jacquemard at LSV-Cachan, C. Godin and P. Barbier de Reuille at CIRAD-Montpellier, Pierre-Etienne Moreau at Loria-Nancy, Éric Goubault at CEA-Saclay, P. Prusinkiewicz at the University of Calgary (who coined the term "computation in space") and the members of the Epigenomic group at GENOPOLE-Évry, for stimulating discussions, thoughtful remarks and warm discussions. We gratefully acknowledge the financial support of the CNRS, the GDR ALP, IMPBIO, the University of Évry and GENOPOLE-Évry.

References

[1] J. Arulanandham, C. Calude, and M. Dinneen. Bead-sort: A natural sorting algorithm. *Bulletin of the European Association for Theoretical Computer Science*, 76:153–162, Feb. 2002. Technical Contributions.

[2] J.-P. Banâtre, A. Coutant, and D. L. Metayer. A parallel machine for multiset transformation and its programming style. *Future Generation Computer Systems*, 4:133–144, 1988.

[3] J.-P. Banâtre, P. Fradet, and D. L. Métayer. Gamma and the chemical reaction model: Fifteen years after. *Lecture Notes in Computer Science*, 2235:17–??, 2001.

[4] G. Berry and G. Boudol. The chemical abstract machine. In *Conf. Record 17th ACM Symp. on Principles of Programmming Languages, POPL'90, San Francisco, CA, USA, 17–19 Jan. 1990*, pages 81–94. ACM Press, New York, 1990.

[5] R. W. Brockett. Dynamical systems that sort lists, diagonalize matrices, and solve linear programming problems. *Linear Algebra and its Applications*, 146:79–91, 1991.

[6] K. M. Chandy. Reasoning about continuous systems. *Science of Computer Programming*, 14(2–3):117–132, Oct. 1990.

[7] E. G. Coffman, M. J. Elphick, and A. Shoshani. System deadlocks. *Computing Surveys*, 3(2):67–78, 1971.

[8] J. Cohen. Typing rule-based transformations over topological collections. In J.-L. Giavitto and P.-E. Moreau, editors, *4th International Workshop on Rule-Based Programming (RULE'03)*, pages 50–66, 2003.

[9] J. Cohen. Typage fort et typage souple des collections topologiques et des transformations. In V. Ménissier-Morain, editor, *Journées Francophones des Langages Applicatifs (JFLA 2004)*, pages 37–54. INRIA, 2004.

[10] P. Fradet and D. L. Métayer. Structured Gamma. *Science of Computer Programming*, 31(2–3):263–289, July 1998.

[11] F. Geurts. Hierarchy of discrete-time dynamical systems, a survey. *Bulletin of the European Association for Theoretical Computer Science*, 57:230–251, Oct. 1995. Surveys and Tutorials.

[12] J.-L. Giavitto. A framework for the recursive definition of data structures. In *ACM-Sigplan 2nd International Conference on Principles and Practice of Declarative Programming (PPDP'00)*, pages 45–55, Montral, Sept. 2000. ACM-press.

[13] J.-L. Giavitto. Invited talk: Topological collections, transformations and their application to the modeling and the simulation of dynamical systems. In *Rewriting Technics and Applications (RTA'03)*, volume LNCS 2706 of *LNCS*, pages 208 – 233, Valencia, June 2003. Springer.

[14] J.-L. Giavitto, C. Godin, O. Michel, and P. Prusinkiewicz. *Modelling and Simulation of biological processes in the context of genomics*, chapter "Computational Models for Integrative and Developmental Biology". Hermes, July 2002. Also republished as an high-level course in the proceedings of the Dieppe spring school on "Modelling and simumation of biological processes in the context of genomics", 12-17 may 2003, Dieppes, France.

[15] J.-L. Giavitto and O. Michel. Declarative definition of group indexed data structures and approximation of their domains. In *Proceedings of the 3nd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP-01)*. ACM Press, Sept. 2001.

[16] J.-L. Giavitto and O. Michel. Mgs: a rule-based programming language for complex objects and collections. In M. van den Brand and R. Verma, editors, *Electronic Notes in Theoretical Computer Science*, volume 59. Elsevier Science Publishers, 2001.

[17] J.-L. Giavitto and O. Michel. `MGS`: a programming language for the transformations of topological collections. Technical Report 61-2001, LaMI – Université d'Évry Val d'Essonne, May 2001.

[18] J.-L. Giavitto and O. Michel. Data structure as topological spaces. In *Proceedings of the 3nd International Conference on Unconventional Models of Computation UMC02*, volume 2509, pages 137–150, Himeji, Japan, Oct. 2002. Lecture Notes in Computer Science.

[19] J.-L. Giavitto and O. Michel. The topological structures of membrane computing. *Fundamenta Informaticae*, 49:107–129, 2002.

[20] J.-L. Giavitto, O. Michel, and J.-P. Sansonnet. Group based fields. In I. Takayasu, R. H. J. Halstead, and C. Queinnec, editors, *Parallel Symbolic Languages and Systems (International Workshop PSLS'95)*, volume 1068 of *LNCS*, pages 209–215, Beaune (France), 2–4 Oct. 1995. Springer-Verlag.

[21] E. Goubault. Geometry and concurrency: A user's guide. *Mathematical Structures in Computer Science*, 10:411–425, 2000.

[22] G.-G. Granger. *La pensée de l'espace*. Odile Jacob, 1999.

[23] M. Henle. *A combinatorial introduction to topology*. Dover publications, New-York, 1994.

[24] J. Jeuring and P. Jansson. Polytypic programming. *Lecture Notes in Computer Science*, 1129:68–114, 1996.

[25] P. Lienhardt. Topological models for boundary representation : a comparison with n-dimensional generalized maps. *Computer-Aided Design*, 23(1):59–82, 1991.

[26] A. Lindenmayer. Mathematical models for cellular interaction in development, Parts I and II. *Journal of Theoretical Biology*, 18:280–315, 1968.

[27] E. Meijer, M. Fokkinga, and R. Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *5th ACM Conference on Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144, Cambridge, MA, August 26–30, 1991. Springer, Berlin.

[28] O. Michel and F. Jacquemard. An analysis of the needham-schroeder public-key protocol with MGS. In G. Mauri, G. Paun, and C. Zandron, editors, *Preproceedings of the Fifth workshop on Membrane Computing (WMC5)*, pages 295–315. EC MolConNet - Universita di Milano-Bicocca, June 2004.

[29] G. Paun. From cells to computers: Computing with membranes (P systems). *Biosystems*, 59(3):139–158, March 2001.

[30] P. Prusinkiewicz and J. Hanan. L systems: from formalism to programming languages. In G. Ronzenberg and A. Salomaa, editors, *Lindenmayer Systems, Impacts on Theoretical Computer Science, Computer Graphics and Developmental Biology*, pages 193–211. Springer Verlag, Feb. 1992.

[31] Z. Róka. One-way cellular automata on Cayley graphs. *Theoretical Computer Science*, 132(1–2):259–290, 26 Sept. 1994.

[32] M. Sintzoff. Invariance and contraction by infinite iterations of relations. In *Research directions in high-level programming languages, LNCS*, volume 574, pages 349–373, Mont Saint-Michel, France, june 1991. Springer-Verlag.

[33] R. D. Sorkin. A finitary substitute for continuous topology. *Int. J. Theor. Phys.*, 30:923–948, 1991.

[34] A. Spicher, O. Michel, and J.-L. Giavitto. A topological framework for the specification and the simulation of discrete dynamical systems. In *Sixth International conference on Cellular Automata for Research and Industry (ACRI'04)*, LNCS, Amsterdam, October 2004. Springer. to be published.

[35] The MAUDE project. Maude home page, 2002. `http://maude.csl.sri.com/`.

[36] The PROTHEO project. Elan home page, 2002. `http://www.loria.fr/equipes/protheo/SOFTWARES/ELAN/`.

[37] H. Weyl. *The Classical Groups (their invariants and representations)*. Princeton University Press, 1939. Reprint edition (October 13, 1997). ISBN 0691057567.

Programming reaction-diffusion processors

Andrew Adamatzky

Faculty of Computing, Engineering and Mathematics,
University of the West of England, UK
andrew.adamatzky@uwe.ac.uk,
http://www.cems.uwe.ac.uk/~aadamatz

Abstract. In reaction (RD) processors, both the data and the results of the computation are encoded as concentration profiles of the reagents. The computation is performed via the spreading and interaction of wave fronts. Most prototypes of RD computers are specialized to solve certain problems, they can not be, in general, re-programmed. In the paper, we try to show possible means of overcoming this drawback. We envisage architecture and interface of programmable RD media capable of solving a wide range of problems.

1 Reaction-diffusion computers

Reaction-diffusion (RD) chemical systems are well known for their unique ability to efficiently solve combinatorial problems with natural parallelism [2]. In liquid-phase parallel processors, both the data and the results of the computation are encoded as concentration profiles of the reagents. The computation *per se* is performed via the spreading and interaction of wave fronts. The RD computers are parallel because the chemical medium's micro-volumes update their states simultaneously, and molecules diffuse and react in parallel (see overviews in [1, 2, 8]). RD information processing in chemical media became a hot topic of not simply theoretical but also experimental investigations since implementation of basic operations of image processing using the light-sensitive Belousov-Zhabotinsky (BZ) reaction [28]. During the last decade a wide range of experimental and simulated prototypes of RD computing devices have been fabricated and applied to solve various problems of computer science, including image processing [35, 3], path planning [43, 12, 34, 6], robot navigation [7, 10], computational geometry [5], logical gates [45, 39, 4], counting [24], memory units [30]. Despite promising preliminary results in RD computing, the field still remains art rather then science, most RD processors are produced on an *ad hoc* basis without structured top-down approaches, mathematical verification, rigorous methodology, relevance to other domains of advanced computing. There is a need to develop a coherent theoretical foundation of RD computing in chemical media. Particular attention should be paid to issues of programmability, because by making RD processors programmable we will transform them from marginal outcasts and curious freaks to enabled competitors of conventional architectures and devices.

2 How to program reaction-diffusion computers?

Controllability is inherent constituent of programmability. How do real chemical media respond to changes in physical conditions? Are they controllable? If yes then what properties of the media can be used most effectively to program these chemical systems? Despite the fact that the problem of controlling RD media did not receive proper attention until recently some preliminary although rather mosaic results have become accessible in the last decade. There is no coherent view on the subject and this will be a major future task to build a theoretical and experimental framework of chemical medium controllability. Below we provide an overview of the findings related to the external control of chemical media. They demonstrate viability of our ideas and show that the present state-of-the-art laboratory methods allow for the precise tuning of these chemical systems, and thus offer an opportunity to program RD processors.

The majority of the literature, related to theoretical and experimental studies concerning the controllability of RD medium, deals with application of an electric field. In a thin-layer BZ reactor stimulated by an electric field the following phenomena are observed: (a) the velocity of excitation waves is increased by a negative and decreased by a positive electric field; (b) a wave is split into two waves that move in opposite directions if a very high electric field is applied across the evolving medium [40]; (c) crescent waves are formed not commonly observed in the field absent evolution of the BZ reaction [23]; (d) stabilisation and destabilisation of wave fronts [26]; (e) an alternating electric field generates a spiral wave core that travels within the medium; the trajectory of the core depends on the field frequency and amplitude [38]. Computer simulations with the BZ medium confirm that (a) waves do not exist in a field-free medium but emerge when a negative field is applied [33]; (b) an electric field causes the formation of waves that change their sign with a change in concentration, and applied constant field induces drift of vortices [32]; (c) externally applied currents cause the drift of spiral excitation patterns [42]. It is also demonstrated that by applying stationary two-dimensional fields to a RD system one can obtain induced heterogeneity in a RD system and thus increase the morphological diversity of the generated patterns (see e.g. [18]). These findings seem to be universal and valid for all RD systems: (a) applying a negative field accelerates wave fronts; (b) increasing the applied positive field causes wave deceleration, wave front retardation, and eventually wave front annihilation; (c) recurrent application of an electric field leads to formation of complex spatial patterns [41]. A system of methylene blue, sulfide, sulfite and oxygen in a polyacrylamide gel matrix gives us a brilliant example of electric-field controlled medium. Typically hexagon and strip patterns are observed in the medium. Application of an electric field makes striped patterns dominate in the medium, even orientation of the stripes is determined by the intensity of the electric field [31].

Temperature is a key factor in the parameterisation of the space-time dynamics of RD media. It is shown that temperature is a bifurcation parameter in a closed non-stirred BZ reactor [29]. By increasing the temperature of the reactor one can drive the space-time dynamic of the reactor from periodic oscillations ($0-3^o$C) to quasi-periodic oscillations ($4-6^o$C) to chaotic oscillations ($7-8^o$C). Similar findings are reported in simulation experiments on discrete media [2], where a lattice node's sensitivity can be considered as an analogue of temperature.

Modifications of reagent concentrations and structure of physical substrate may indeed contribute to shaping space-time dynamics of RD media. Thus, by varying the concentration of malonic acid in a BZ medium one can achieve (a) the formation of target waves; (b) the annihilation of wave fronts; and, (c) the generation of stable propagating reduction fronts [26]. By changing substrate we can achieve transitions between various types of patterns formed, see e.g. [22] on transitions between hexagons and stripes. This however could not be accomplished 'on-line', during the execution of a computational process, or even between two tasks, the whole computing device should be 're-fabricated', so we do not consider this option prospective. Convection is yet another useful factor governing space-time dynamics of RD media. Thus, e.g., convection 2nd order waves, generated in collisions of excitation waves in BZ medium, may travel across the medium and affect, e.g. annihilate, existing sources of the wave generation [36].

Light was the first [27] and still remains the best, see overview in [35], way of controlling spatio-temporal dynamics of RD media (this clearly applies mostly to light-sensitive species as BZ reaction). Thus, applying light of varying intensity we can control medium's excitability [19] and excitation dynamic in BZ-medium [17, 25], wave velocity [37], and patter formation [46]. Of particular interest to implementation of programmable logical circuits are experimental evidences of light-induced back propagating waves, wave-front splitting and phase shifting [47].

3 Three examples of programming RD processors

In this section we briefly demonstrate a concept of control-based programmability in models of RD processors. Firstly, we show how to adjust reaction rates in RD medium to make it perform computation of Voronoi diagram over a set of given points. Secondly, we provide a toy model of tunable three-valued logical gates, and show how to re-program a simple excitable gate to implement several logical operations by simply changing excitability of the medium's sites. Thirdly, we indicate how to implement logical circuits in architecture-less RD excitable medium.

Consider a cellular automaton model of an abstract RD excitable medium. Let a cell x of two-dimensional lattice takes four states: resting $\circ$, excited $(+)$, refractory and precipitated $\star$, and update their states in discrete time t depending on a number $\sigma^t(x)$ of excited neighbors in its eight-cell neighborhood as follows. Resting cell x becomes excited if $0 < \sigma^t(x) \leq \theta_2$ and precipitated if $\theta_2 < \sigma^t(x)$. Excited cell 'precipitates' if $\theta_1 < \sigma^t(x)$ and becomes refractory otherwise. Refractory cell recovers to resting state unconditionally, and precipitate cell does not change its state. Initially we perturb medium, excite it in several sites, thus inputting data. Waves of excitation are generated, they grow, collide with each other and annihilate in result of the collision. They may form a stationary inactive concentration profile of a precipitate, which represents result of the computation. Thus, we can only be concerned with reactions of precipitation: $+ \rightarrow^{k_1} \star$ and $\circ \boxplus + \rightarrow^{k_2} \star$, where k_1 and k_2 are inversely proportional to θ_1 and θ_2, respectively. Varying θ_1 and θ_2 from 1 to 8, and thus changing precipitation rates from maximum possible to a minimum one, we obtain various kinds of precipitate patterns, as shown in Fig. 1. Precipitate patterns developed for rel-

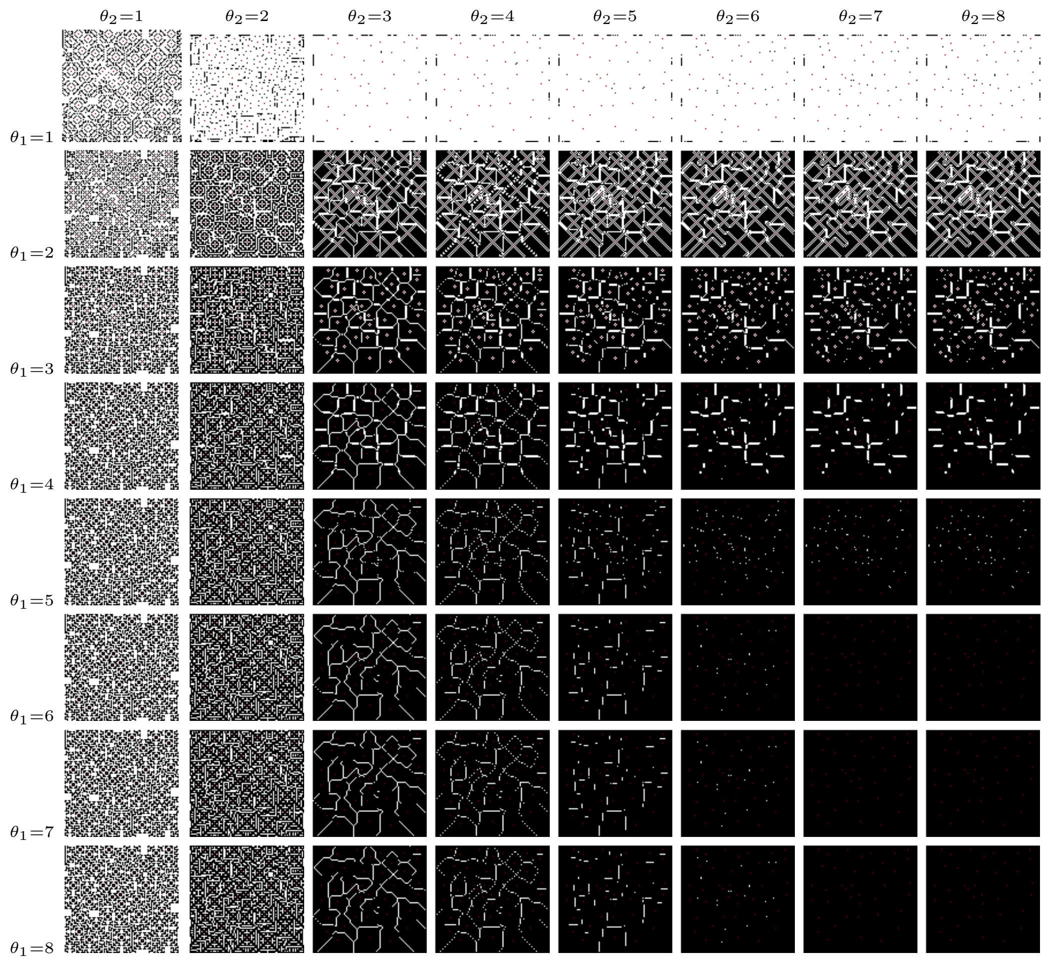


Fig. 1. Final configurations of RD medium for $1 \leq \theta_1 \leq \theta_2 \leq 2$. Resting sites are black, precipitate is white.

atively high ranges of reactions rates: $3 \leq \theta_1, \theta_2 \leq 4$ represent discrete Voronoi diagrams (given 'planar' set, represented by sites of initial excitation, is visible in pattern $\theta_1 = \theta_2 = 3$ as white dots inside Voronoi cells) derived from the set of initially excited sites. This example demonstrates that externally controlling precipitation rates we can force RD medium to compute Voronoi diagram.

Consider a T-shaped excitable RD medium built of three one-dimensional cellular arrays joined at one point (details are explained in [11]); two channels are considered as inputs, and third channel as an output. Every cell of this structure has two neighbors apart of end cells, which have one neighbor each, and a junction cell, which has three neighbors. Each cell takes three states: resting ($\circ$), excited ($+$) and refractory ($-$). A cell switches from excited state to refractory state, and from refractory to resting unconditionally. If resting cell excites when certain amount of its neighbors is excited then waves of excitation, in the form $+-$, travel along the channels of the gate. Waves generated in input channels, meet at a junction, and may pass or not pass to the output channel. We represent logical values as follows: no waves is FALSE, one wave $+-$ is NONSENSE and two waves $+ - \cdot + -$ represent TRUTH. Assume that sites of the excitable gate are highly excitable: every cell excites if at least one neighbor is excited. One or two waves generated at one of the inputs pass onto output channel; two single waves are merged in one single wave when collide at the junction; and, a single wave is 'absorbed' by train of two waves. Therefore, the gate with highly excitable sites implements Łukasiewicz disjunction (Fig. 2a). Let us decrease sites sensitivity

$\vee_Ł$	T	F	$\star$
T	T	T	T
F	T	F	$\star$
$\star$	T	$\star$	$\star$

(a)

$\wedge_Ł$	T	F	$\star$
T	T	F	$\star$
F	F	F	F
$\star$	$\star$	F	$\star$

(b)

$\boxdot$	T	F	$\star$
T	F	T	$\star$
F	T	F	$\star$
$\star$	$\star$	$\star$	F

(c)

Fig. 2. Operations of Łukasiewics three-valued logic implemented in models of T-shaped excitable gate: (a) disjunction, (b) conjunction, (c) NOT-EQUIVALENCE gate.

and make it depend on number k of cell neighbors: a cell excites if at least $\lceil \frac{k}{2} \rceil$ neighbors are excited. Then junction site can excite only when exactly two of its neighbors are excited, therefore, excitation spreads to output channels only when two waves meet at the junction. Therefore, when a single wave collide to a train of two waves the only single wave passes onto output channel. In such conditions of low excitability the gate implements Łukasiewicz conjunction (Fig. 2b). By further narrowing excitation interval: a cell is excited if exactly one neighbor is excited, we achieve situation when two colliding wave fronts annihilate, and thus output channel is excited only if either of input channels is excited, or if the input channels got different number of waves. Thus, we implement combination of Łukasiewicz NOT and EQUIVALENCE gates (Fig. 2c).

Logical circuits can be also fabricated in uniform, architecture-less, where not wires or channels are physically implemented, excitable RD medium, (e.g. sub-excitable BZ medium as numerically demonstrated in [9]) by generation, reflection and collision of traveling wave fragments. To study the medium we integrate two-variable Oregonator equation, adapted to a light-sensitive BZ reaction with applied illumination [17]

$$\frac{\partial u}{\partial t} = \frac{1}{\epsilon}(u - u^2 - (fv + \phi)\frac{u-q}{u+q}) + D_u \nabla^2 u$$

$$\frac{\partial v}{\partial t} = u - v$$

where variables u and v represent local concentrations of bromous acid and oxidized catalyst ruthenium, ϵ is a ratio of time scale of variables u and v, q is a scaling parameter depending on reaction rates, f is a stoichiometric coefficient, ϕ is a light-induced bromide production rate proportional to intensity of illumination. The system supports propagation of sustained wave fragments, which may be used as representations of logical variables (e.g. absence is FALSE, presence is TRUTH). To program the medium we should design initial configuration of perturbations, that will cause excitation waves, and configurations of deflectors and prisms, to route these quasi-particle wave-fragments. While implementation of Boolean operations *per se* is relatively straightforward [9], control of signal propagation, routing and multimplication of signals is of most importance when considering circuits not simply singe gates. To multiply a signal or to change wave-fragment trajectory we can temporarily apply illumination impurities to change local properties of the medium on a way the wave. Thus we can cause the wave-fragment to split (Fig. 3ab) or deflect (Fig. 3cd). A control impurity (Fig. 3bd), or deflector, consists of a few segments of sites which illumination level is slightly above or below overall illumination level of the medium. Combining these excitatory and inhibitory segments we can precisely control wave's trajectory, e.g. realize U-turn of a signal (Fig. 3cd).

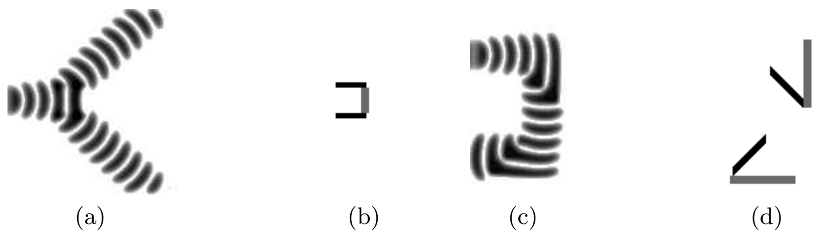


Fig. 3. Operating wave fragments. Overlay of images taken every 0.5 time units. Exciting domains of impurities are shown in black, inhibiting domains are gray. (a) Signal branching with impurity: wave fragment traveling west is split by impurity (b) into two waves traveling north-west and south-west. (d) Signal routing (U-turn) with impurities: wave fragment traveling east is routed north and then west by two impurities (e).

4 Discussion

Sluggishness, narrow range of computational tasks solved, and seeming unsusceptibility to a control are usually seen as main disadvantages of existing prototypes of RD computers. In the paper we briefly outlined several ways of external controlling, tuning, and ultimately programming, of spatially extended chemical devices. We have also indicated how to 'switch' a RD computer, with fixed set of reactions but variable reaction rates, between several domains of problems, and thus make it more 'omnivorous'. Thus we made grounds for partial dismissal of specialization and uncontrollability statements. As to the speed, real-life RD processors are slow indeed, due to limitations on speed of diffusion and phase waves traveling in a liquid layer or a gel. We can argue, however, that future applications of the chemical processors lie in the field of micro-scale computing devices and soft robotic architectures, e.g. gel-made robots, where RD medium forms an integral part of robot body [44]. A silicon fabrication is another way, however possibly a step back from material implementation point of view, to improve speed of RD computers. This route seems to be well developed, particularly in designing RD media in non-linear circuits and cellular neural networks [20, 21, 10]. CMOS design and analog emulation of RD systems, BZ medium in particular, have already demonstrated feasibility of mapping chemical dynamics onto silicon architectures [13–16]. Semiconductor devices based on minor carrier transport [16], like arrays of *p*-*n*-*p*-*n* diod based reaction devices, give us a hope for forthcoming designs of nano-scale RD processors.

Acknowledgement

I thank Ben De Lacy Costello for numerous discussions on laboratory prototypes of reaction-diffusion computers, Tetsuya Asai for enlightening me in electronic reaction-diffusion devices, and participants of the EPSRC Cluster in Novel Computation Workshop "Non-Linear Media Based Computers" (Bristol, September, 2003) for help in shaping my ideas. I also thank Peter Dittrich for encouraging me to think about programmability of reaction-diffusion processors.

References

1. Adamatzky, A.: Reaction-diffusion and excitable processors: a sense of the unconventional. Parallel and Distributed Computing **3** (2000) 113–132.
2. Adamatzky, A.: Computing in Nonlinear Media and Automata Collectives (Institute of Physics Publishing, 2001).
3. Adamatzky, A., De Lacy Costello, B. and Ratcliffe, N.M.: Experimental reaction-diffusion pre-processor for shape recognition. Physics Letters A **297** (2002) 344–352.
4. Adamatzky, A. and De Lacy Costello, B.P.J.: Experimental logical gates in a reaction-diffusion medium: The XOR gate and beyond. Phys. Rev. E **66** (2002) 046112.

5. Adamatzky, A. and De Lacy Costello, B.P.J.: On some limitations of reaction-diffusion computers in relation to Voronoi diagram and its inversion. Physics Letters A **309** (2003) 397–406.
6. Adamatzky, A. and De Lacy Costello, B.P.J.: Reaction-diffusion path planning in a hybrid chemical and cellular-automaton processor. Chaos, Solitons & Fractals **16** (2003) 727–736.
7. Adamatzky, A., De Lacy Costello, B., Melhuish, C. and Ratcliffe, N.: Experimental reaction-diffusion chemical processors for robot path planning. J. Intelligent & Robotic Systems **37** (2003) 233–249.
8. Adamatzky, A.: Computing with waves in chemical media: massively parallel reaction-diffusion processors. IEICE Trans. (2004), in press.
9. Adamatzky, A.: Collision-based computing in BelousovZhabotinsky medium. Chaos, Solitons & Fractals **21** (2004) 1259–1264.
10. Adamatzky, A., Arena, P., Basile, A., Carmona-Galan, R., De Lacy Costello, B., Fortuna, L., Frasca, M., Rodriguez-Vazquez, A.: Reaction-diffusion navigation robot control: from chemical to VLSI analogic processors. IEEE Trans. Circuits and Systems I, **51** (2004) 926–938.
11. Adamatzky, A. and Motoike, I.: Three-valued logic gates in excitable media, in preparation (2004).
12. Agladze, K., Magome, N., Aliev, R., Yamaguchi, T. and Yoshikawa, K.: Finding the optimal path with the aid of chemical wave. Physica D **106** (1997) 247–254.
13. Asai, T., Kato, H., and Amemiya, Y.: Analog CMOS implementation of diffusive Lotka-Volterra neural networks, INNS-IEEE Int. Joint Conf. on Neural Networks, P-90, Washington DC, USA, July 15–19, 2001.
14. Asai, T., Nishimiya, Y. and Amemiya, Y.: A CMOS reaction-diffusion circuit based on cellular-automaton processing emulating the Belousov-Zhabotinsky reaction. IEICE Trans. on Fundamentals of Electronics, Communications and Computer, **E85-A** (2002) 2093–2096.
15. Asai, T. and Amemiya, Y.: Biomorphic analog circuits based on reaction-diffusion systems. Proc. 33rd Int. Symp. on Multiple-Valued Logic (Tokyo, Japan, May 16-19, 2003) 197–204.
16. Asai, T., Adamatzky, A., Amemiya, Y.: Towards reaction-diffusion semiconductor computing devices based on minority-carrier transport. Chaos, Solitons & Fractals **20** (2004) 863–876.
17. Beato, V., Engel, H.: Pulse propagation in a model for the photosensitive Belousov-Zhabotinsky reaction with external noise. In: Noise in Complex Systems and Stochastic Dynamics, Edited by Schimansky-Geier, L., Abbott, D., Neiman, A., Van den Broeck, C. Proc. SPIE **5114** (2003) 353–62.
18. Bouzat, S. and Wio, H.S.: Pattern dynamics in inhomogeneous active media. Physica A **293** (2001) 405–420.
19. Brandtstädter, H., Braune, M., Schebesch, I. and Engel, H.: Experimental study of the dynamics of spiral pairs in light-sensitive BelousovZhabotinskii media using an open-gel reactor. Chem. Phys. Lett. **323** (2000) 145–154.
20. Chua, L.O.: CNN: A Paradigm for Complexity (World Scientific Publishing, 1998).
21. Chua, L.O. and Roska, T.: Cellular Neural Networks and Visual Computing: Foundations and Applications (Cambridge University Press, 2003).
22. De Kepper, P., Dulos, E., Boissonade, J., De Wit, A., Dewel, G. and Borckmans, P.: Reaction-diffusion patterns in confined chemical systems. J. Stat. Phys. **101** (2000) 495–508.

23. Feeney, R., Schmidt, S.L. and Ortoleva, P.: Experiments of electric field-BZ chemical wave interactions: annihilation and the crescent wave. Physica D **2** (1981) 536–544.
24. Gorecki, J., Yoshikawa, K., Igarashi, Y.: On chemical reactors that can count. J. Phys. Chem. A **107** (2003) 1664–1669.
25. Grill, S., Zykov, V. S., Müller, S. C.: Spiral wave dynamics under pulsatory modulation of excitability. J. Phys. Chem. **100** (1996) 19082–19088.
26. Kastánek P., Kosek, J., Snita,D., Schreiber, I. and Marek, M.: Reduction waves in the BZ reaction: Circles, spirals and effects of electric field, Physica D **84** (1995) 79–94.
27. Kuhnert, L.: Photochemische Manipulation von chemischen Wellen. Naturwissenschaften **76** (1986) 96–97.
28. Kuhnert, L., Agladze, K.L. and Krinsky, V.I.: Image processing using light–sensitive chemical waves. Nature **337** (1989) 244–247.
29. Masia, M., Marchettini, N., Zambranoa, V. and Rustici, M.: Effect of temperature in a closed unstirred Belousov-Zhabotinsky system. Chem. Phys. Lett. **341** (2001) 285–291.
30. Motoike, I.N. and Yoshikawa, K.: Information operations with multiple pulses on an excitable field. Chaos, Solitons & Fractals **17** (2003) 455–461.
31. Muenster, A.F, Watzl, M. and Schneider, F.W.: Two-dimensional Turing-like patterns in the PA-MBO-system and effects of an electric field. Physica Scripta **T67** (1996) 58–62.
32. Mu*ñ*uzuri, A.P., Davydov, V.A., Pérez-Mu*ñ*uzuri, V., Gómez-Gesteira, M. and Pérez-Villar, V.: General properties of the electric-field-induced vortex drift in excitable media. Chaos, Solitons, & Fractals **7** (1996) 585–595.
33. Ortoleva, P.: Chemical wave-electrical field interaction phenomena. Physica D **26** (1987) 67–84.
34. Rambidi, N.G. and Yakovenchuck, D.: Finding path in a labyrinth based on reaction–diffusion media. Adv. Materials for Optics and Electron. **7** (1999) 67–72.
35. Rambidi, N.: Chemical-based computing and problems of high computational complexity: The reaction-diffusion paradigm, In: Seinko, T., Adamatzky, A., Rambidi, N., Conrad, M., Editors, Molecular Computing (The MIT Press, 2003).
36. Sakurai, T., Miike, H., Yokoyama, E. and Muller, S.C.: Initiation front and annihilation center of convection waves developing in spiral structures of Belousov-Zhabotinsky reaction. J. Phys. Soc. Japan **66** (1997) 518–521.
37. Schebesch, I., Engel, H.: Wave propagation in heterogeneous excitable media. Phys. Rev. E **57** (1998) 3905-3910.
38. Seipel, M., Schneider, F.W. and Mnster, A.F.: Control and coupling of spiral waves in excitable media. Faraday Discussions **120** (2001) 395–405.
39. Sielewiesiuka, J. and Górecki, J.: On the response of simple reactors to regular trains of pulses. Phys. Chem. Chem. Phys. **4** (2002) 1326-1333.
40. Sevćikova, H. and Marek, M.: Chemical waves in electric field. Physica D **9** (1983) 140–156.
41. Sevćikova, H. and Marek, M.: Chemical front waves in an electric field. Physica D **13** (1984) 379–386.
42. Steinbock O., Schutze J., Muller, S.C.: Electric-field-induced drift and deformation of spiral waves in an excitable medium. Phys. Rev. Lett. **68** (1992) 248–251.
43. Steinbock, O., Tóth, A. and Showalter, K.: Navigating complex labyrinths: optimal paths from chemical waves. Science **267** (1995) 868–871.

44. Tabata, O., Hirasawa, H., Aoki, S., Yoshida, R. and Kokufuta, E.: Ciliary motion actuator using selfoscillating gel. Sensors and Actuators A **95** (2002) 234–238.
45. Tóth, A. and Showalter, K.: Logic gates in excitable media. J. Chem. Phys. **103** (1995) 2058–2066.
46. Wang, J.: Light-induced pattern formation in the excitable Belousov-Zhabotinsky medium. Chem. Phys. Lett. **339** (2001) 357–361.
47. Yoneyama, M.: Optical modification of wave dynamics in a surface layer of the Mn-catalyzed Belousov-Zhabotinsky reaction. Chem. Phys. Lett. **254** (1996) 191–196.

From Prescriptive Programming of Solid-state Devices to Orchestrated Self-organisation of Informed Matter

Klaus-Peter Zauner

School of Electronics and Computer Science
University of Southampton
SO17 1BJ, United Kingdom
kpz@ecs.soton.ac.uk

Abstract. Achieving real-time response to complex, ambiguous, high-bandwidth data is impractical with conventional programming. Only the narrow class of compressible input-output maps can be specified with feasibly sized programs. Efficient physical realizations are embarrassed by the need to implement the rigidly specified instructions requisite for programmable systems. The conventional paradigm of erecting stern constraints and potential barriers that narrowly prescribe structure and precisely control system state needs to be complemented with a new approach that relinquishes detailed control and reckons with autonomous building blocks. Brittle prescriptive control will need to be replaced with resilient self-organisation to approach the robustness and efficency afforded by natural systems.
Self-organising processes ranging from self-assembly to growth and development will play a central role in mimicking the high integration density of nature's information processing architectures in artificial devices. Structure-function self-consistency will be key to the spontaneous generation of functional architectures that can harness novel molecular and nano materials in an effective way for increased computational power.

1 Commanding the Quasi-universal Machine

The common conventional computer is an approximation of a hypothetical universal machine [1] limited by memory and speed constraints. Universal machines are generally believed to be in principle able to compute any effectively computable function [2]. Correspondingly it is assumed that if processing speed and memory space of computers would indefinitely continue to increase, any computable information processing problem would eventually come within reach of practical devices. Accordingly time and space complexity of computation has been studied in detail [3] and technological advances have focused on memory capacity and switching speed [4]. But along with this there is another factor that limits realizable computing devices: the length of the program required to

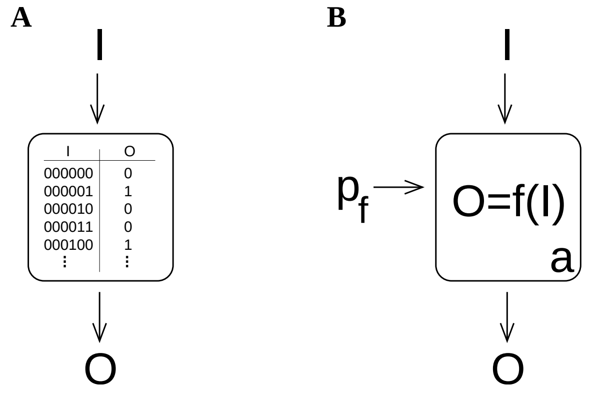


Fig. 1. Communicating a desired input-output map to a machine. The input-output map can in principle be thought of as a potentially very large lookup table that associates an output response with every input that can be discerned by the machine (**A**). For n bit input patterns I and a m bit output (O) response the, number of possible maps is 2^{m2^n}. To implement an arbitrary one of these maps on a quasi-universal machine, the mapping f has to be specified by the program p with respect of machine architecture a (**B**). Selecting an arbitrary map from the set of possible maps may require a specification of length: $\log_2 \left[2^{m2^n}\right] = m2^n$. Even for moderate pattern recognition problems (e.g., classifying low resolution images) the program length required for most mappings is impractical [6].

communicate a desired behaviour to the device [5]. The length of this program is limited by the state space of the device and the capacity of the programmers. Both limits can be exhausted rather quickly (cf. figure 1). As a consequence conventional computing architectures are in practice restricted to the implementation of highly compressible input-output maps [7]. The set of compressible maps is a small subset of the potential input-output functions—most behaviours cannot be programmed. Whether the incompressible and thus inaccessible mappings are useful is an open question. In the light of the ability of organisms to cope with complex ambiguous pattern recognition problems it appears likely that input-output mappings of limited compressibility can be valuable in practice.

The picture painted so far is too optimistic. It assumes that the machine architecture is not degenerate, i.e., no two different programmes implement the same input-output map. In practice, however, the mapping of input into output is achieved by decomposing the transformation into a series of sequential elementary information processing operations. Information processing is essentially selective dissipation of information and each operation entails a, possibly delayed, loss of information [8, 9]. Now if the transformation of input signal patterns is decomposed into a large number of operations, all information pertaining to the input may be dissipated in the processing sequence. And so it may happen that the output response will be independent of the input and thus constant [10, 11]. This further reduces the number of input-output maps accessible through programs.

For a machine to be programmable, additional restrictions come into play. Programming is here equated with an engineering approach in which mental conception precedes physical creation (cf. [12]). It necessitates the possibility for the programmer to anticipate the actions of the available elementary operations. Only if the function of the elementary operations can be foreseen by the programmer a desired input-output map can be implemented by incrementally composing a program. Accordingly, the machine's architecture has to adhere to a fixed, finite user manual to facilitate programming. To achieve this, numerous potential interactions among the components of the machine need to be suppressed [13]. Programmability is achieved by using relatively large networks of components with fixed behaviour. This however does not allow for the efficiency afforded by networks of context sensitive components [14].

As outlined above, conventional programming is not always the most suitable form of implementing an input-output map. Some maps cannot be compressed into programs of practical length, and the need for programmability precludes hardware designs that elicit functionality from a minimum of material.

2 Learning, Adaptation, Self-organisation

Programmability is not a prerequisite for the realization of information processing systems as is exemplified by the enviable computing capabilities of cells and organisms. Artificial neural networks provide a technological example of non-programmed information processing [15]. They trade an effective loss of programmability for parallel operation. Freeing the computing architecture from the need for predictable function of elementary components opens up new design degrees of freedom. Firstly, the fan-in for an elementary component could be increased by orders of magnitude. It may be interesting to note that neurons in the cortex of the mouse have on average 8000 input lines [16]. Secondly, there is no need for all components to operate according to identical specifications. This opens a path to broadening the material basis of computation by allowing for computational substrates the structure of which cannot be controlled in detail. And likewise, thirdly, the operation of the elementary components can be depended on their context in the architecture, thus greatly increasing the number of interactions among the components that can be recruited for signal fusion.

Utilising these degrees of freedom will abrogate present training algorithms for artificial neural networks. At the same time however, the increased dimensionality enhances the evolvability of such networks. The evolutionary paradigm of using the performance of an existing system as an estimate for the expected performance of an arbitrarily modified version of the system can cope with the complexity and inhomogeneity of architectures based on context sensitive components. In fact it is particularly effective in this domain [17].

3 Orchestrating Informed Matter

Techniques for producing biomaterials and manufacturing nano-materials are rapidly developing. In the very near future we will see materials with unprecedented characteristics arriving at an increasing rate. But there is nothing to indicate that we are on a path to harnessing these new materials for increased computational power. Drilling the materials to act as logic gates is unlikely to be fruitful. Present computing concepts enforce formalisms that are arbitrary from the perspective of the physics underlying their implementation.

Nature's gadgets process information in starkly different ways than conventionally programmed machines do [18]. They exploit the physics of the materials directly and arrive at problem solutions driven by free energy minimisation while current computer systems are coerced by high potential barriers to follow a narrowly prescribed, contrived course of computation. The practical implementation of an input-output map can adhere in varying degrees to different paradigms as illustrated in figure 2. Selecting functional structures from a pool of randomly created structures is particularly suitable for nano-materials where detailed control is not feasible or not economical. If the process of structure formation is repeatable then the selection from random variation can be iterated for evolutionary progress. In general, however, evolving highly complex input-output maps from scratch may be impractical. It is here where the concept of *informed matter* [19], i.e., molecules deliberately designed to carry information that enables them to interact individually, autonomously with other molecules, comes into play. Combining the abstract concepts of artifical chemistry [20] with the physics of supramolecular chemistry [21, 22] conceivably will enable the orchestration of self-organisation to arrive in practical time scales at physics driven architectures.

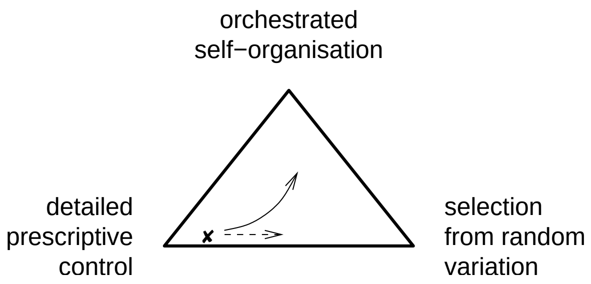

orchestrated self−organisation
detailed prescriptive control
selection from random variation

Fig. 2. Implementation paradigms for a computational device. Present conventional computer technology is indicated near the lower left corner. Random variation enters unintentionally in the production process. With increasing miniaturisation control will become increasingly more difficult (dashed arrow). Resilient architectures that can cope with wide component variation and the deliberate use of self-organisation processes provide the most likely path to complexification of computing architectures (bent arrow).

4 Perspectives

A likely early application niche for the principles outlined in the previous section is the area of autonomous micro robotic devices. With the quest for robots at a scale of a cubic millimetre and below molecular controllers become increasingly attractive [23, 24], and initial steps towards implementation are underway [25]. Coherent perception-action under real-time constraints with severely limited computational resources does not allow for the inefficiency of a virtual machine that abstracts physics away. For satisfactory performance the robot's control needs to adapt directly to the reality of its own body [26]. In fact the body structure can be an integral part of the computational infrastructure [27, 28].

About 18 million organic compounds are known today—a negligible number if compared to the space of possible organic molecules, estimated to 10^{63} substances [29]. Nature offers a glimpse at what is available in this space of possibilities with organised, adaptive, living, thinking matter. Following Lehn's trail-blazing call to "ultimately acquire the ability to create new forms of complex matter" [19] will require information processing paradigms tailored to the microphysics of the underlying computational substrate.

References

1. Turing, A.M.: On computable numbers with an application to the entscheidungsproblem. In: Proceedings of the London Mathematical Society. Volume 42. (1937) 230–265 Corrections, Ibid vol. 43 (1937), pp. 544–546. Reprinted in *The Undecideable*, M. Davis, ed., Raven Press, New York, 1965.
2. Minsky, M.L.: Computation: Finite and Infinite Machines. Prentice-Hall, Englewood Cliffs, N.J. (1967)
3. Papadimitriou, C.H.: Computational Complexity. Addison-Wesley, Reading (1994)
4. Compañó, R.: Trends in nanoelectronics. Nanotechnology **12** (2001) 85–88
5. Chaitin, G.J.: On the length of programs for computing finite binary sequences. J. Assoc. Comput. Mach. **13** (1966) 547–569
6. Conrad, M., Zauner, K.P.: Conformation-based computing: a rational and a recipe. In Sienko, T., Adamatzky, A., Rambidi, N., Conrad, M., eds.: Molecular Computing. MIT Press, Cambridge, MA (2003) 1–31
7. Zauner, K.P., Conrad, M.: Molecular approach to informal computing. Soft Computing **5** (2001) 39–44
8. Landauer, R.: Irreversibility and heat generation in the computing process. IBM Journal **5** (1961) 183–191
9. Landauer, R.: Fundamental limitations in the computational process. Berichte der Bunsen-Gesellschaft **80** (1976) 1048–1059
10. Langdon, W.B.: How many good programs are there? How long are they? In et. al., J.R., ed.: Foundations of Genetic Algorithms FOGA-7, Torremolinos, 4-6 September, Morgan Kaufmann (2002)
11. Langdon, W.B.: The distribution of reversible functions is normal. In Riolo, R., ed.: Genetic Programming Theory and Practice, Ann Arbor, 15–17 May, Proceedings, Dordrecht, Kluwer Academic Publishers (2003)

12. Pfaffmann, J.O., Zauner, K.P.: Scouting context-sensitive components. In Keymeulen, D., Stoica, A., Lohn, J., Zebulum, R.S., eds.: The Third NASA/DoD Workshop on Evolvable Hardware—EH-2001, Long Beach, 12–14 July 2001, IEEE Computer Society, Los Alamitos (2001) 14–20
13. Conrad, M.: Scaling of efficiency in programmable and non-programmable systems. BioSystems **35** (1995) 161–166
14. Conrad, M.: The price of programmability. In Herken, R., ed.: The Universal Turing Machine: A Fifty Year Survey. Oxford University Press, New York (1988) 285–307
15. Partridge, D.: Non-programmed computation. Communications of the ACM **43** (2000) 293–302
16. Schüz, A.: Neuroanatomy in a computational perspective. In Arbib, M.A., ed.: The Handbook of Brain Theory and Neural Networks. MIT Press, Cambridge, MA (1995) 622–626
17. Miller, J.F., Downing, K.: Evolution in materio: Looking beyond the silicon box. In: 2002 NASA/DoD Conference on Evolvable Hardware (EH'02), July 15 - 18, 2002, Alexandria, Virginia, IEEE (2002) 167–176
18. Conrad, M.: Information processing in molecular systems. Currents in Modern Biology (now BioSystems) **5** (1972) 1–14
19. Lehn, J.M.: Supramolecular chemistry: from molecular information towards self-organization and complex matter. Reports on Progress in Physics **67** (2004) 249–265
20. Dittrich, P., Ziegler, J., Banzhaf, W.: Artificial chemistries—a review. Artificial Life **7** (2001) 225–275
21. Lehn, J.M.: Supramolecular chemistry—scope and perspectives: Molecules, supermolecules and molecular devices. Angewandte Chemie, Int. Ed. Engl. **27** (1988) 90–112
22. Whiteside, G.M., Mathias, J.P., Seto, C.T.: Molecular self-assembly and nanochemistry: A chemical startegy for the synthesis of nanostructures. Science **254** (1991) 1312–1319
23. Ziegler, J., Dittrich, P., Banzhaf, W.: Towards a metabolic robot controller. In Holcombe, M., Paton, R., eds.: Information Processing in Cells and Tissues. Plenum Press, New York (1998)
24. Adamatzky, A., Melhuish, C.: Parallel controllers for decentralized robots: towards nano design. Kybernetes **29** (2000) 733–745
25. Adamatzky, A., de Lacy Costello, B., Melluish, C., Ratcliffe, N.: Experimental implementation of mobile robot taxis with onboard belousov-zhabotinsky chemical medium. Materials Science & Engineering C (2004) To appear.
26. Elliott, T., Shadbolt, N.R.: Developmental robotics: manifesto and application. Phil. Trans. R. Soc. Lond. A **361** (2003) 2187–2206
27. Hasslacher, B., Tilden, M.W.: Living machines. Robotics and Autonomous Systems (1995) 143–169
28. Rietman, E.A., Tilden, M.W., Askenazi, M.: Analog computation with rings of quasiperiodic oscillators: the microdynamics of cognition in living machines. Robotics and Autonomous Systems **45** (2003) 249–263
29. Scheidtmann, J., Weiß, P.A., Maier, W.F.: Hunting for better catalysts and materials—combinatorial chemistry and high throughput technology. Applied Catalysis A: General **222** (2001) 79–89

Relational growth grammars – a graph rewriting approach to dynamical systems with a dynamical structure

Winfried Kurth[1], Ole Kniemeyer[1], and Gerhard Buck-Sorlin[12]

[1] Brandenburgische Technische Universität Cottbus, Department of Computer Science, Chair for Practical Computer Science / Graphics Systems, P.O.Box 101344, 03013 Cottbus, Germany
{wk, okn}@informatik.tu-cottbus.de
[2] Institute of Plant Genetics and Crop Plant Research (IPK), Dept. Cytogenetics, Corrensstr. 3, 06466 Gatersleben, Germany
buck@ipk-gatersleben.de

1 Introduction

Rule-based programming is one of the traditionally acknowledged paradigms of programming [2], but its application was in most cases restricted to logical inference or to spaces with restricted forms of topology: Grids in the case of cellular automata [12], locally 1-dimensional branched structures in the case of classical L-systems [8]. Recently, there is growing interest in rule-based simulation of dynamical systems with a dynamical structure, using diverse types of data structures and topologies, motivated by biological and chemical applications [4].

In this paper we propose a rewriting formalism, "relational growth grammars" (RGG), acting on relational structures, i.e., graphs with attributed edges, which generalizes L-systems and allows the specification of dynamical processes on changing structures in a wide field of applications. By associating the nodes of the graphs with classes in the sense of the object-oriented paradigm, we gain further flexibility. Here we will explain the fundamental concepts and show simple examples to demonstrate the essential ideas and possibilities of this approach.

Our work was motivated by the demand for a uniform modelling framework capable to represent genomes and macroscopic structures of higher organisms (plants) in the same language [6]. In the context of a "virtual crops" project, we thus created the language XL, an extension of Java allowing a direct specification of RGGs, and the software GroIMP (Growth-Grammar related Interactive Modelling Platform) enabling interpretation of XL code and easy user interaction during the simulations [5].—This research was funded by the DFG under grant Ku847/5-1 and additionally supported by IPK. All support is gratefully acknowledged.

2 Relational growth grammars

An RGG rule is a quintuple (L, C, E, R, P) with $L \cup C \neq \emptyset$. L, the *left-hand side proper* of the rule, is a set of graphs with node labels and edge labels. A *derivation step* of an RGG involves the removal of a copy ("match") of one rule's L from a (usually) larger graph and the insertion of the corresponding R, the *right-hand side proper* of the rule, which is also a set of graphs (with the underlying node sets not necessarily disjunct from those of L). C is again a set of labelled graphs (with the node set possibly but not necessarily overlapping with that of L) and is called the *context* of the rule. For a rule, in order to be applicable the set C must match with a set of subgraphs of the given graph in a way which is consistent with the match of L, but in the derivation step the context is not removed (except for the parts that are also in L). This notion of context generalizes the "left" and "right contexts" of context-sensitive L-systems [8] and enables a flexible control of subgraph replacement by specifying a local situation which must be given before a rule can be applied. E is a set of logical expressions in some syntax which we are not going to define in detail here. These expressions usually contain some parameters referring to node labels from $L \cup C$ and are interpreted as *conditions* which must be met before the rule can be applied. Conditions may also contain some function calls evoking a random number generator—if this is the case the rule is called *stochastic*. Finally, P is a (possibly empty) list of commands which are also not specified syntactically here, possibly involving parameters referring to node labels from $L \cup C \cup R$ and parameters from E. P specifies a procedural piece of code which is executed after rule application.—We write RGG rules in the form

```
(* C *), L, (E) ==> R{P};
```

the arrangement of the C, L and E parts not being fixed.

Figure 1 illustrates a simple example of an RGG rule with $C = E = P = \emptyset$ (upper part) and its application to a graph (lower part). A possible text notation for this rule in our XL syntax would be `i -b-> j, j -a-> k, k -a-> i  ==> j` .

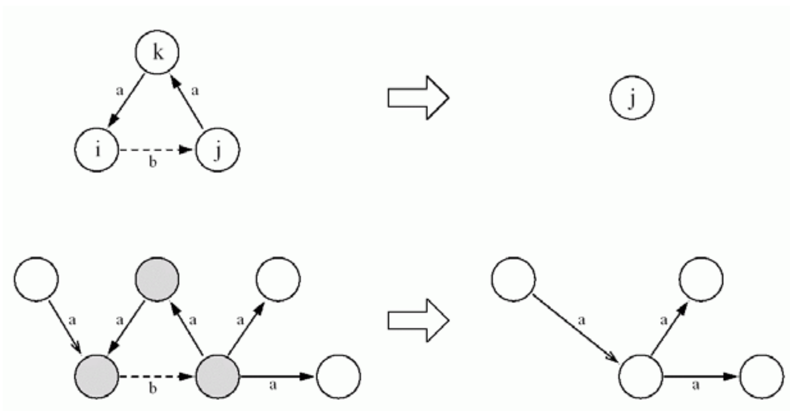


Fig. 1. An RGG rule in graphical notation (upper part) and its application to a given graph (lower part). The shaded nodes are matched by the left-hand side of the rule. (From [6].)

In our applications, we found it useful to have a short notation for rules with $L = R = \varnothing$. In this case, only parameters from the context C (and possibly from E) appear in the procedural part, and we write

$$C\ \texttt{,}\ \texttt{(}E\texttt{)}\ \ \texttt{::>}\ P\,.$$

A RGG is a set of RGG rules. In the language XL, RGG rules can be put together in blocks, thus enabling an additional hierarchy of rules and an explicit control of their order of application, like in table L-systems [9]. An RGG-based *derivation* is a sequence of discrete, successive derivation steps, starting from a given initial graph (axiom). In each step, one or all matching rules are applied, depending on the chosen mode of rule application (see below).

RGGs were partly inspired by the PROGRES graph grammar system [11]. Furthermore, features from parametric L-systems were incorporated into the RGG formalism. Particularly, commands of turtle geometry (cf. [8]) are allowed as nodes and can serve to interpret the derived graphs geometrically.

Pure rule-based programming does not in all situations allow an intuitive access. The notion of "programmed graph replacement systems" [10] was introduced to overcome this limitation: Additional programming structures, following a different paradigm, are supported. In the RGG approach, the inclusion of P in our definition allows the execution of code from a conventional object-oriented language. Additionally, in XL such a language (Java) serves as a framework for the whole RGG and allows the user to define constants, variables, classes and methods. Furthermore, graph nodes in XL are Java objects and can carry arbitrary additional information and functionalities, e.g., concerning geometry, visual appearance or animated behaviour.

3 Key features of RGGs

Two issues require special consideration in graph grammars: The *mode of rule application* and the *embedding* of the right-hand side into the graph immediately after rule application. *Sequential* and *parallel* mode of application are well known from Chomsky grammars and L-systems, respectively. In most physical, chemical and biological applications, the parallel mode turns out to be more appropriate. However, the parallel application of rules requires the specification of a *conflict resolution strategy* for overlapping matches of left-hand sides. Future extensions of the RGG formalism will contain explicit support for the most common conflict resolution schemes. Until now, we have considered only a special case: the multiple matching of L with one and the same copy of L in the graph. This occurs always when L allows some automorphism and when C and E do not enforce a selection between the matches. The standard mode of rule application realized in XL, which is basically the single-pushout approach (also known as the algebraic or Berliner approach) [1], tries to apply the rule to every match. In many applications it is more meaningful to apply a rule only once to an *underlying node set* of a match. (The selection among the isomorphic matches has then to be done either nondeterministically or following a specified strategy.) This option will be implemented in a later XL version; currently it must be emulated by additional conditions in part E of the rule.

Our standard mechanism of *embedding* simply transfers incoming (outgoing) edges of the textually leftmost (rightmost) nodes of L to the textually leftmost (rightmost) nodes of R. Future versions of XL will allow other embedding strategies.

We have shown in another paper [6] that it is straightforward to represent typical *standard data structures* like sets, multisets, lists or multiple-scaled trees as labelled graphs. The RGG formalism provides standard types of edges (i.e., special edge labels) to represent common relations occurring in these data structures, like the successor relation in lists or the membership relation in sets. Because the successor relation is used so often, it is denoted by a blank in our notation, i.e., `a b` is equivalent to `a -successor-> b`. Additionally, the user can define new relations, using e.g. algebraic operators like the transitive hull, which can be used in RGG rules in the same way as edges.

4 Example 1: Spreading of a signal in a network

We assume that a signal jumps in each time step from a cell to all adjacent cells of a network and changes the state of the reached cells from "inactive" (0) to "active" (1). In our implementation in XL we make use of the possibility to use objects as nodes which can carry arbitrary additional data, in this case the state of the cell. "Cell" is a predefined class which contains a variable called "state", but such a class could also be defined by the user. The signal propagation requires only one RGG rule:

```
(* c1: Cell *) c2: Cell, (c1.state == 1) ==> c2(1).
```

Here, the blank between `*)` and `c2` denotes the successor relation in the network where the rule is to be applied. "`Cell`" is the node type (class), "`c1`" and "`c2`" are labels of nodes serving to distinguish them and to refer to them on the right-hand side, and "`state`" is a variable of "`Cell`" which can likewise be seen as a parameter of the nodes and which is forced to 1 at the node `c2` on the right-hand side. The way in which the context is specified in this rule is a short notation for the definition-conforming rule

```
(* c1: Cell c2: Cell *), c2, (c1.state == 1) ==> c2(1).
```

The result of two steps of rule application is illustrated in Fig. 2.

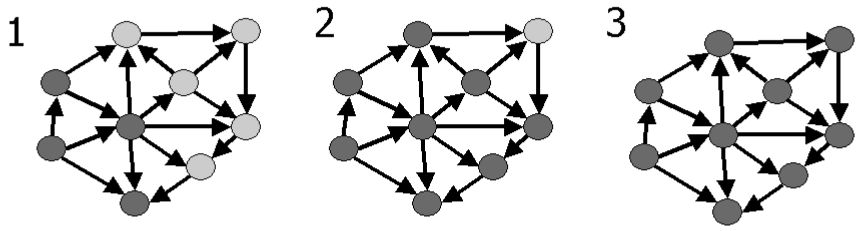


Fig. 2. Application of the signal propagation rule to a network. Dark nodes have state 1, light-grey nodes 0.

This example simulates the dynamics of some state variable, but there is no dynamical structure; the topology of the network remains unchanged. Hence this is a simple case, but using the same basic rule structure it is possible to simulate the dynamics of complex metabolic networks and gene regulation networks, using real-valued concentrations and Michaelis-Menten kinetics in the state-changing rules (see [6] for an example).

5 Example 2: "Game of Life"

Cellular automata (CA) can easily be expressed as RGGs. This is demonstrated at the example of the "Game of Life", a well-known 2-dimensional CA with nontrivial longterm behaviour [3]. We use again the class "Cell", with state value 1 for "living" and 0 for "dead". In the following piece of XL code, we have omitted only the initialization of the grid, which can be done by a trivial rule creating cells at predefined positions.

```
boolean neighbour(Cell c1, Cell c2)
  { return (c1 != c2) && (c1.distanceLinf(c2) < 1.1); }
public void run()
  [
  x:Cell(1),
    (!(sum( (* x -neighbour-> #Cell *).state) in {2..3}))
       ==> x(0);
  x:Cell(0),
    (  sum( (* x -neighbour-> #Cell *).state) == 3)
       ==> x(1);
  ]
```

The first declaration defines the Moore neighbourhood as a new edge type between "Cell" nodes, based upon a geometric relation (chessboard distance lower than 1.1). The block "run" contains the transition function of the CA in the form of two RGG rules. These rules cover only the cases where the state switches from 0 to 1 or vice versa; no rule is necessary to force a cell *not* to change its state. The conditions in both rules make use of an *arithmetical-structural operator*, `sum`, which was first introduced in the context of L-systems [7]. Its argument is iteratively evaluated for all nodes matching with the node marked by `#` in the context specification `(*` ... `*)` and added up.—Figure 3 demonstrates the possibility of user interaction during RGG execution, which is provided by the GroIMP software. (a) and (b) show successive steps in the undisturbed development of a "Game of Life" configuration (living cells are black and enlarged). After every step, the user can stop the RGG derivation process and interfere manually; e.g., he may change the state of a cell. In (c), a cell was even removed from its position and placed at an irregular location on the grid. Rule application can nevertheless be resumed, leading to a "disturbed" dynamics (d).

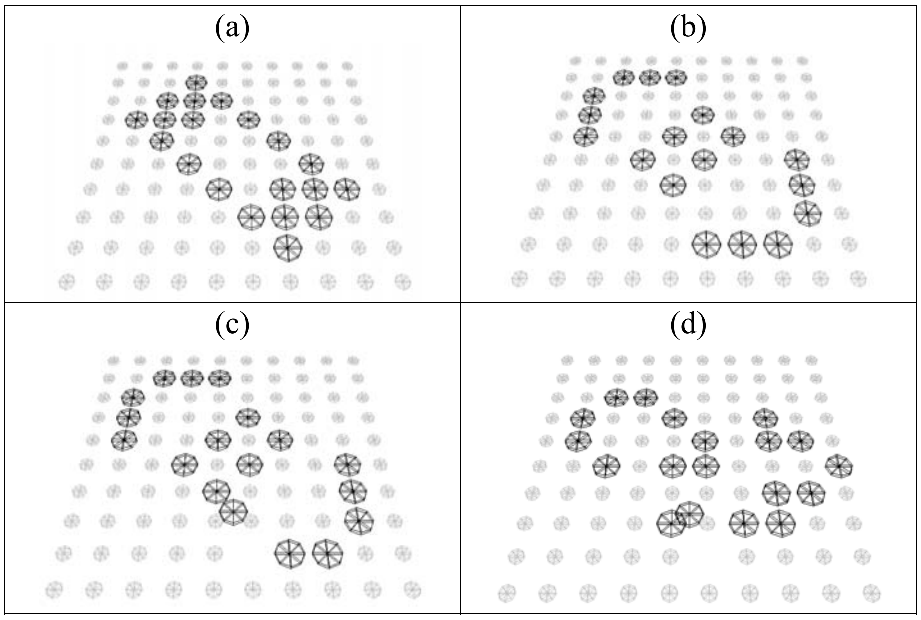


Fig. 3. A symmetric configuration in the Game of Life (a) and its successor configuration (b). In (c), one "living" cell of (b) was translated by the user. The disturbed configuration develops asymmetrically, shown in (d) after several steps.

In this example, structural change can only be obtained by an intervention of the user. The next example shows the growth of a geometrical structure.

6 Example 3: A distance-sensitive plant

The following rules simulate the growth of a plant which consists of a main axis, ending in an apical meristem `m`, and short lateral branches (emanating from invisible nodes of type "`s`") with the capacity of flowering. The apical bud of such a branch is transformed into a visible flower only if there is no other bud or flower closer to the bud than a given threshold distance. The corresponding RGG is similar to a classical L-system, but the global character of the sensitivity involved in the distance criterion excludes a realization as a "context-sensitive" L-system (with "context" interpreted with respect to strings).—We omit again the initialization rule.

```
module m(int x) extends Sphere(3);
module s;
module inflor;
module bud extends inflor;
(...)
m(x)  ==> F(12) if (x>0) ( RH(180) [s] m(x-1) );
s     ==> RU(irandom(50, 70)) F(irandom(15, 18)) bud;
```

```
b:bud ==> inflor
          if (forall(distance(b,(* #x:inflor,(b!=x)*))
                     > 13)
             ( RL(70) [ F(4) RH(50)
                      for (1..5) ( RH(72)
                                 [ RL(80) F(3) ]) ] );
```

The system uses the turtle commands **F** (creation of a cylinder) and **RH**, **RL**, **RU** (spatial rotations), cf. [8, 7]. The second rule is stochastic and introduces some randomness in the branching angles and branch lengths. The third rule makes again use of arithmetical-structural operators (**forall**, **distance**) and iterative evaluation of a set defined by a context definition. Note that the class hierarchy specified in the module declarations is used to count not only inflorescences, but also buds as potentially flower-inhibiting objects (**bud extends inflor**). Furthermore, conditional and loop structures (**if**, **for**) are used to organize the right-hand side. The "**for**" construction generalizes the repetition operator used in L-systems [7].— Figure 4 shows two stages of a derivation which started with three "m" nodes (meristems). Flowering turns out to be partially inhibited for the central plant, due to a lack of free space.

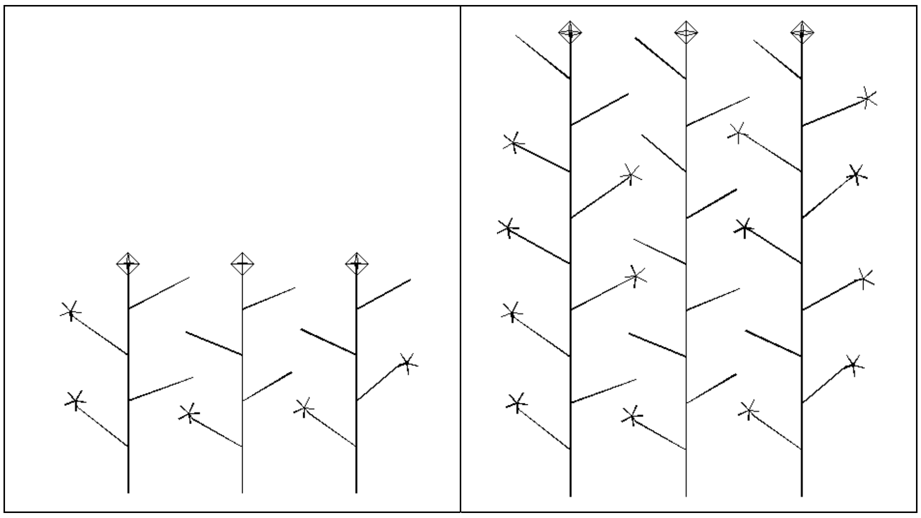


Fig. 4. Growth of a "sensitive" plant with flowering restricted to conditions where a distance threshold to the nearest competing object must be surpassed.

Our last example combines structural changes with dynamic, physically-based simulation of state variables. It shows also the applicability of RGGs in the field of chemistry.

7 Example 4: A polymerization model with mass-spring kinetics

Spherical "monomers" are enclosed in a rectangular region and move in random initial directions with constant velocity. The region boundaries are reflecting. Each time two monomers have a close encounter, a chemical bond is formed between them which is modelled in analogy to a spring. Impulse is conserved in the resulting dimer. Reactions resulting in a binding can also happen between yet unsaturated ends of a polymer. It is assumed that each monomer has only two binding sites. Thus the resulting polymers have either a chain or a ring topology (see Fig. 5).

The RGG specifying this behaviour consists of not more than 2 module definitions ("Monomer" and "Spring") and 4 rules—one for initialization, one for constant movement with reflection at the walls, one for spring mechanics and one for chemical binding. This RGG will be documented in detail in a forthcoming version of this paper.

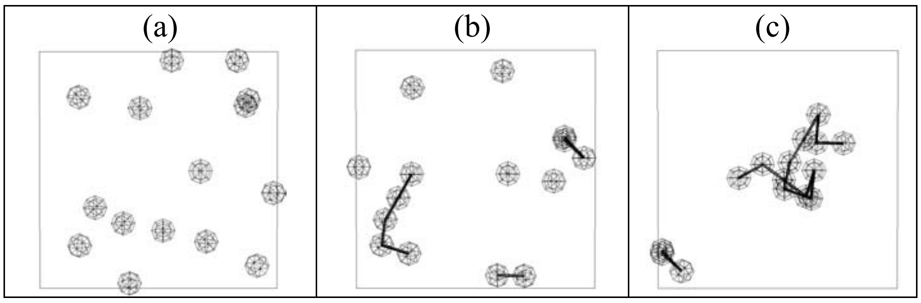


Fig. 5. Development simulated by the polymerisation model. (a) initial situation, (b) after several collisions, two chains and one ring (of three monomers, one of them being occluded in the picture) have formed, (c) finally one large chain and the small ring remain.

8 Conclusions

Relational growth grammars permit the writing of short and comprehensible model specifications for a wide field of scientific applications, ranging from physical and chemical simulations (Example 4) to the growth of plants (Example 3), genetic processes and metabolic networks [6]. They are essentially rule-based, but because they operate on unordered *sets* of graphs, they share also some characteristics with chemical computing. Furthermore, in the language XL they are implemented in a way that enables straightforward combination with procedural and object-oriented constructions as well. Hence we see an important potential for future applications of this flexible approach.

References

1. Ehrig, H., Korff, M., Löwe, M., Ribeiro, L., Wagner, A., Corradini, A.: Algebraic approaches to graph transformation—part II: Single pushout approach and comparison with double pushout approach. In: G. Rozenberg (ed.): Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1, Foundations. World Scientific, Singapore (1997) 247–312.
2. Floyd, R.: The paradigms of programming. Communications of the ACM 22 (1979) 455–460.
3. Gardner, M.: Wheels, Life, and Other Mathematical Amusements. W.H. Freeman, New York (1983).
4. Giavitto, J.-L., Michel, O.: MGS: A rule-based programming language for complex objects and collections. Electronic Notes in Theoretical Computer Science 59 (4) (2001).
5. Kniemeyer, O.: Rule-based modelling with the XL/GroIMP software. In: H. Schaub, F. Detje, U. Brüggemann (eds.): The Logic of Artificial Life. Proceedings of 6th GWAL, April 14–16, 2004, Bamberg, Germany. AKA, Berlin (2004) 56–65.
6. Kniemeyer, O., Buck-Sorlin, G., Kurth, W.: A graph grammar approach to Artificial Life. Artificial Life 10 (4) (2004) 413–431.
7. Kurth, W.: Some new formalisms for modelling the interactions between plant architecture, competition and carbon allocation. Bayreuther Forum Ökologie 52 (1998) 53–98.
8. Prusinkiewicz, P., Lindenmayer, A.: The Algorithmic Beauty of Plants. Springer, New York (1990).
9. Rozenberg, G.: T0L systems and languages. Information and Control 23 (1973) 357–381.
10. Schürr, A.: Programmed graph replacement systems. In: G. Rozenberg (ed.): Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1, Foundations. World Scientific, Singapore (1997) 479–546.
11. Schürr, A., Winter, A.J., Zündorf, A.: The PROGRES approach: Language and environment. In: G. Rozenberg (ed.): Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2, Applications, Languages and Tools. World Scientific, Singapore (1999) 487–550.
12. Wolfram, S.: Cellular Automata and Complexity. Collected papers. Addison-Wesley, Reading (1994).

A new programming paradigm inspired by Artificial Chemistries

W. Banzhaf[1] and C. Lasarczyk[2]

[1] Department of Computer Science, Memorial University of Newfoundland, St. John's NL A1B 3X5, CANADA
[2] Department of Computer Science, University of Dortmund, D–44221 Dortmund, GERMANY

Abstract. In this contribution we shall introduce a new method of program execution, based on notions of Artificial Chemistries. Instead of executing instructions in a predefined sequential order, execution will be in random order in analogy to chemical reactions happening between substances. It turns out that such a model of program execution is able to achieve desirable goals if augmented by an automatic program searching method like Genetic Programming. We demonstrate the principle of this approach and discuss prospects and consequences for parallel exectution of such programs.

1 Introduction

In this contribution we shall introduce a new way of looking at transformations from input to output different from a prescribed sequence of computational steps. Instead, the elements of the transformation, which in our case are single instructions from a multiset $I = \{I_1, I_2, I_3, I_2, I_3, I_1, ...\}$ are drawn in a random order to produce a transformation result. In this way we dissolve the sequential order usually associated with an algorithm for our programs. It will turn out, that such an arrangement is still able to produce wished-for results, though only under the reign of a programming method that banks on its stochastic character. This method will be Genetic Programming.

A program in this sense is thus not a sequence of instructions but rather an assemblage of instructions that can be executed in arbitrary order. By randomly choosing one instruction at a time, the program proceeds through its transformations until a predetermined number of instructions has been executed. In the present work we set the number of instructions to be executed at five times the size of the multiset, this way giving ample chance to each instruction to be executed at least once and to exert its proper influence on the result.

Different multi-sets can be considered different programs, whereas different passes through a multi–set can be considered different behavioral variants of a single program. Programs of this type can be seen as artificial chemistries, where instructions interact with each other (by taking the transformation results from one instruction and feeding it into another). As it will turn out, many interactions of this type are, what in an Artificial Chemistry is called "elastic", in that nothing happens as a result, for instance because the earlier instruction did not feed into the arguments of the later.[3]

Because instructions are drawn randomly in the execution of the program, it is really the concentration of instructions that matters most. It is thus expected that "programming" of such a system requires the proper choice of concentrations of instructions, similar to what is required from the functioning of living cells, where at each given time many reactions happen simultaneously but without a need to synchronicity.

2 Background

Algorithmic Chemistries were considered earlier in the work of Fontana [9]. In that work, a system of λ-calculus expressions was examined in their interaction with each other. Due to the nature of the λ-calculus, each expression could serve both as a function and as an argument to a function. The resulting system produced, upon encounter of λ-expressions, new λ-expressions.

In our contribution we use the term as an umbrella term for those kinds of artificial chemistries [7] that aim at algorithms. As opposed to terms like randomized or probabilistic algorithms, in which a certain degree of stochasticity is introduced explicitely, our algorithms have an implicit type of stochasticity. Executing the sequence of instructions every time in a different order has the potential of producing highly unpredictable results.

It will turn out, however, that even though the resulting computation is unpredictable in principle, evolution will favor those multi-sets of instructions that turn out to produce approximately correct results after execution. This feature of approximating the wished-for results is a consequence of the evolutionary forces of mutation, recombination and selection, and will have nothing to do with the actual order in which instructions are being executed. Irrespective of how many processors would work on the multi-set, the results of the computation would tend to fall into the same band of approximation. We submit, therefore, that methods like this can be very useful in parallel and distributed environements.

Our previous work on Artificial Chemistries (see, for example [2, 5, 6, 12]) didn't address the question of how to write algorithms "chemically" in enough detail. In [3] we introduced a very general analogy between chemical reaction and algorithmic computation, arguing that concentrations of results would be important. The present contribution aims to fill that gap and to put forward a proposal as to how such an artificial chemistry could look like.

3 The Method

Genetic Programming (GP) [10, 4] belongs to the family of Evolutionary Algorithms (EA). These heuristic algorithms try to improve originally random

[3] Elastic interactions have some bearings on neutral code, but they are not identical.

solutions to a problem via the mechanisms of recombination, mutation and selection. Many applications of GP can be described as evolution of models [8]. The elements of models are usually arithmetic expressions, logical expressions or executable programs.

Here, we shall use evolution of approximation and classification problems to demonstrate the feasibility of the approach. We represent a program as a set of instructions only stored as a linear sequence in memory due to technical limitations. These instructions are 2 and 3 address instructions which work on a set of registers.

3.1 Linear GP with Sequence generators

Here we shall use 3-address machine instructions. The genotype of an individual is a list of those instructions. Each instruction consists of an operation, a destination register, and two source registers[4]. Initially, individuals are produced by randomly choosing instructions. As is usual, we employ a set of fitness cases in order to evaluate (and subsequently select) individuals.

Figure 1 shows the execution of an individual in linear GP. A sequence gen-

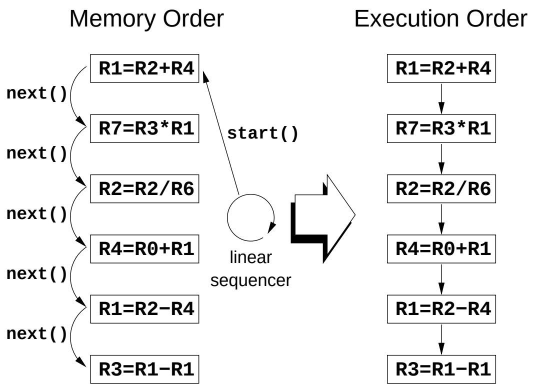


Fig. 1. Execution of an individual in linear GP. Memory order and execution order correspond to each other. Arrows indicate returned values of calls to the sequence generator.

erator is used to determine the sequence of instructions. Each instruction is executed, with resulting data stored in its destination register. Usually, the sequence generator moves through the program sequence instruction by instruction. Thus, the location in memory space determines the particular sequence of instructions. Classically, this is realized by the program counter.[5]

[4] Operations which require only one source register simply ignore the second register.
[5] (Conditional) jumps are a deviation from this behavior.

1–Point–*Crossover* can be described using two sequence generators. The first generator is acting on the first parent and returns instructions at its beginning. These instructions form the first part of the offspring. The second sequence generator operates on the other parent. We ignore the first instructions this generator returns[6]. The others form the tail of the offsprings instruction list.

Mutation changes single instructions by changing either operation, or destination register or the source registers according to a prescribed probability distribution.

3.2 A register machine as an Algorithmic Chemistry

There is a simple way to realize an chemistry by a register machine. By substituting the systematic incremental stepping of the sequence generator by a random sequence we arrive at our system. That is to say, the instructions are drawn randomly from the set of all instructions in the program[7]. Still, we have to provide the number of registers, starting conditions and determine a target register from which output is to be drawn.

As shown in Figure 2 the chemistry works by executing the instructions of an individual analogous to what would happen in a linear GP–System (cf. 1), except that the sequence order is different.

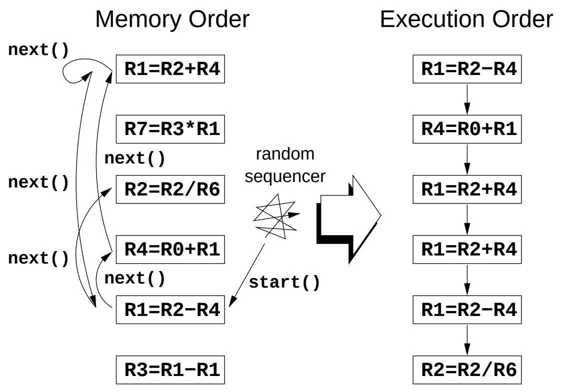


Fig. 2. Execution in the AC system. The sequence generator returns a random order for execution.

It should be noted that there are registers with different features: Some registers are read-only. They can only be used as source registers. These registers contain constant values and are initialized for each fitness case at the start of

[6] Should crossover generate two offspring, the instructions not copied will be used for a second offspring.

[7] For technical reasons instructions are ordered in memory space, but access to an instruction (and subsequent execution) are done in random order.

program execution. All other registers can be read from and written into. These are the connection registers among which information flows in the course of the computation. Initially they are set to zero.

How a program behaves during execution will differ from instance to instance. There is no guarantee that an instruction is executed, nor is it guaranteed that this happens in a definite order or frequency. If, however, an instruction is more frequent in the multi-set, then its execution will be more probable. Similarly, if it should be advantageous to keep independence between data paths, the corresponding registers should be different in such a way that the instructions are not connecting to each other. Both features would be expected to be subject to evolutionary forces.

3.3 Evolution of an Algorithmic Chemistry

Genetic programming of this algorithmic chemistry (ACGP) is similar to other GP variants. The use of a sequence generator should help understand this similarity. We have seen already in Section 3.2 how an individual in ACGP is evaluated.

Initialization and mutation Initialization and mutation of an individual are the same for both the ACGP and usual linear GP.

Mutation will change operator and register numbers according to a probability distribution. In the present implementation register values are changed using a Gaussian with mean at present value and standard deviation 1.

Crossover Crossover makes use of the randomized sequences produced by the sequence generator. As shown in Figure 3 a random sequence of instructions is copied from the parents to the offspring. Though the instructions inherited

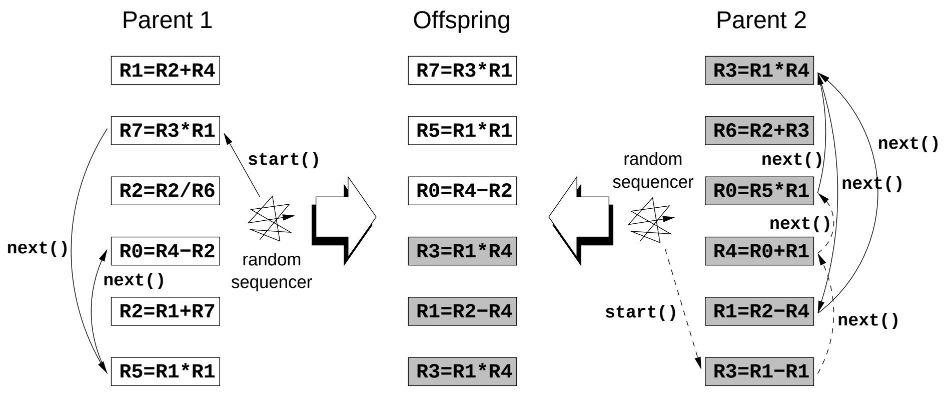


Fig. 3. Crossover in an Artificial Chemistry.

from each of the parents are located in contiguous memory locations, the actual sequence of the execution is not dependent on that order. The probability that a particular instruction is copied into an offspring depends on the frequency of that instruction in the parent. Inheritance therefore is inheritance of frequencies of instructions, rather than of particular sequences of instructions.

Constant register values will be copied with equal probability from each parent, as is done for choice of the result register.

4 Results and Outlook

Using examples from regression and classification we show that goal-oriented behaviour is possible with a seemingly uncoordinated structure of program elements.

The similarity of this approach to dataflow architectures [1] is obvious. Traditional restrictions of that architecture, however, can be loosened with the present model of non-deterministic computation, "programmed" by evolution. Recent work in the dataflow community [11] might therefore find support in such an approach.

The strength of this approach will only appear if distributedness is taken into account. The reasoning would be the following: Systems of this kind should consist of a large number of processing elements which would share program storage and register content. Elements would asynchroneously access storage and register. The program's genome wouldn't specify an order for the execution of instructions. Instead, each element would randomly pick instructions and execute them. Communication with the external world would be performed via a simple control unit.

It goes without saying that such a system would be well suited for parallel processing. Each additional processing element would accelerate the evaluation of programs. There would be no need for massive communication and for synchronization between processing elements. The system would be scalable at run-time: New elements could be added or removed without administrative overhead. The system as a whole would be fault-tolerant, failure of processing elements would appear merely as a slowed-down execution. Loss of information would not be a problem, and new processes need not be started instead of lost ones. Reducing the number of processors (and thus slowing down computation) could be allowed even for power management.

Explicit scheduling of tasks would not be necessary. Two algorithmic chemistries executing different tasks could be unified into one even, provided they used different connection registers. Would it be necessary that one task should be prioritized a higher concentration of instructions would be sufficient to achieve that.

Acknowledgement

The authors gratefully acknowledge support from a grant of the Deutsche Forschungsgemeinschaft DFG to W.B. under Ba 1042/7–3.

References

1. Arwind, and Kathail, V. A multiple processor data flow machine that supports generalized procedures. In *International Conference on Computer Architecture (Minneapolis 1981)* (Los Alamitos, CA, 1981), IEEE Computer Society.
2. Banzhaf, W. Self-replicating sequences of binary numbers. *Comput. Math. Appl. 26* (1993), 1–8.
3. Banzhaf, W. Self-organizing Algorithms Derived from RNA Interactions. In *Evolution and Biocomputing*, W. Banzhaf and F. Eeckman, Eds., vol. 899 of *LNCS*. Springer, Berlin, 1995, pp. 69–103.
4. Banzhaf, W., Nordin, P., Keller, R., and Francone, F. *Genetic Programming - An Introduction*. Morgan Kaufmann, San Francisco, CA, 1998.
5. di Fenizio, P. S., Dittrich, P., Banzhaf, W., and Ziegler, J. Towards a Theory of Organizations. In *Proceedings of the German 5th Workshop on Artificial Life* (Bayreuth, Germany, 2000), M. Hauhs and H. Lange, Eds., Bayreuth University Press.
6. Dittrich, P., and Banzhaf, W. Self-Evolution in a Constructive Binary String System. *Artificial Life 4*, 2 (1998), 203–220.
7. Dittrich, P., Ziegler, J., and Banzhaf, W. Artificial Chemistries - A Review. *Artificial Life 7* (2001), 225–275.
8. Eiben, G., and Smith, J. *Introduction to Evolutionary Computing*. Springer, Berlin, Germany, 2003.
9. Fontana, W. Algorithmic chemistry. In *Artificial Life II* (Redwood City, CA, 1992), C. G. Langton, C. Taylor, J. D. Farmer, and S. Rasmussen, Eds., Addison-Wesley, pp. 159–210.
10. Koza, J. *Genetic Programming*. MIT Press, Cambridge, MA, 1992.
11. Swanson, S., Michelson, K., and Oskin, M. Wavescalar. Tech. rep., University of Washington, Dept. of Computer Science and Engineering, 2003.
12. Ziegler, J., and Banzhaf, W. Evolving Control Metabolisms for a Robot. *Artificial Life 7* (2001), 171–190.

Higher-order Chemical Programming Style

J.-P. Banâtre[1], P. Fradet[2] and Y. Radenac[1]

[1] IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France
(jbanatre,yradenac)@irisa.fr
[2] INRIA Rhône-Alpes, 655 avenue de l'Europe, 38330 Montbonnot, France
Pascal.Fradet@inria.fr

Abstract. The chemical reaction metaphor describes computation in terms of a chemical solution in which molecules interact freely according to reaction rules. Chemical solutions are represented by multisets of elements and computation proceeds by consuming and producing new elements according to reaction conditions and transformation rules. The chemical programming style allows to write many programs in a very elegant way. We go one step further by extending the model so that rewrite rules are themselves molecules. This higher-order extension leads to a programming style where the implementation of new features amounts to adding new active molecules in the solution representing the system.

1 Introduction

The chemical reaction metaphor has been discussed in various occasions in the literature. This metaphor describes computation in terms of a chemical solution in which molecules (representing data) interact freely according to reaction rules. Chemical solutions are represented by multisets. Computation proceeds by rewritings of the multiset which consume and produce new elements according to reaction conditions and transformation rules.

To the best of our knowledge, the Gamma formalism was the first "chemical model of computation" proposed as early as in 1986 [1] and later extended in [2]. A Gamma program is a collection of reaction rules acting on a multiset of basic elements. A reaction rule is made of a condition and an action. Execution proceeds by replacing elements satisfying the reaction condition by the elements specified by the action. The result of a Gamma program is obtained when a stable state is reached, that is to say, when no reaction can take place any more. Figure 1 gives three short examples illustrating the Gamma style of programming. The

$$\begin{aligned} max &= \textbf{replace}\ x, y\ \textbf{by}\ x\ \ \textbf{if}\ x > y \\ primes &= \textbf{replace}\ x, y\ \textbf{by}\ y\ \ \textbf{if}\ multiple(x, y) \\ maj &= \textbf{replace}\ x, y\ \textbf{by}\{\}\ \textbf{if}\ x \neq y \end{aligned}$$

Fig. 1. Examples of Gamma programs

reaction *max* computes the maximum element of a non empty set. The reaction replaces any couple of elements x and y such that $x > y$ by x. This process goes on till a stable state is reached, that is to say, when only the maximum element remains. The reaction *primes* computes the prime numbers lower or equal to a given number N when applied to the multiset of all numbers between 2 and N ($multiple(x, y)$ is true if and only if x is multiple of y). The majority element of a multiset is an element which occurs more than $card(M)/2$ times in the multiset. Assuming that such an element exists, the reaction *maj* yields a multiset which only contains instances of the majority element just by removing pairs of distinct elements. Let us emphasize the conciseness and elegance of these programs. Nothing had to be said about the order of evaluation of the reactions. If several disjoint pairs of elements satisfy the condition, the reactions can be performed in parallel.

Gamma makes it possible to express programs without artificial sequentiality. By artificial, we mean sequentiality only imposed by the computation model and unrelated to the logic of the program. This allows the programmer to describe programs in a very abstract way. In some sense, one can say that Gamma programs express the very idea of an algorithm without any unnecessary linguistic idiosyncrasies. The interested reader may find in [2] a long series of examples (string processing problems, graph problems, geometry problems, ...) illustrating the Gamma style of programming and in [3] a review of contributions related to the chemical reaction model.

This article presents a higher-order extension of the Gamma model where all the computing units are considered as molecules reacting in a solution. In particular, reaction rules are molecules which can react or be manipulated as any other molecules. In Section 2, we exhibit a minimal higher-order chemical calculus, called the γ-calculus, which expresses the very essence of chemical models. This calculus is then enriched with conditional reactions and the possibility of rewriting atomically several molecules. This model suggests a programming style where the implementation of new features amounts to adding new active molecules in the solution representing the system. Section 3 illustrates the characteristics of the model through the example of an autonomic mail system. Section 4 concludes and suggests several research directions.

2 A minimal chemical calculus

In this section, we introduce a higher-order calculus, the γ_0-*calculus* [4], that can be seen as a formal and minimal basis for the chemical paradigm (in much the same way as the λ-calculus is the formal basis of the functional paradigm).

2.1 Syntax and semantics

The fundamental data structure of the γ_0-calculus is the multiset. Computation can be seen either intuitively, as chemical reactions of elements agitated by Brownian motion, or formally, as higher-order associative and commutative (AC) rewritings of multisets. The syntax of γ_0-terms (also called *molecules*) is given in Figure 2. A γ-abstraction is a reactive molecule which consumes a molecule (its

$$\begin{array}{rlll} M ::= & x & ; \textit{variable} \\ \mid & \gamma\langle x\rangle.M & ; \gamma\textit{-abstraction} \\ \mid & M_1, M_2 & ; \textit{multiset} \\ \mid & \langle M\rangle & ; \textit{solution} \end{array}$$

Fig. 2. Syntax of γ_0-molecules

argument) and produces a new one (its body). Molecules are composed using the AC multiset constructor ",". A solution encapsulates molecules (*e.g.,* multiset) and keeps them separate. It serves to control and isolate reactions.

The γ_0-calculus bears clear similarities with the λ-calculus. They both rely on the notions of (free and bound) variable, abstraction and application. A λ-abstraction and a γ-abstraction both specify a higher-order rewrite rule. However, λ-terms are tree-like whereas the AC nature of the application operator "," makes γ_0-terms multiset-like. Associativity and commutativity formalizes Brownian motion and make the notion of solution necessary, if only to distinguish between a function and its argument.

The conversion rules and the reduction rule of the γ_0-calculus are gathered in Figure 3. Chemical reactions are represented by a single rewrite rule, the γ-

$$\begin{array}{lcll l} (\gamma\langle x\rangle.M), \langle N\rangle & \longrightarrow_\gamma & M[x := N] & \text{if } \mathit{Inert}(N) \vee \mathit{Hidden}(x, M) & ; \gamma\textit{-reduction} \\ \gamma\langle x\rangle.M & \equiv & \gamma\langle y\rangle.M[x := y] & \text{with } y \text{ fresh} & ; \alpha\textit{-conversion} \\ M_1, M_2 & \equiv & M_2, M_1 & & ; \textit{commutativity} \\ M_1, (M_2, M_3) & \equiv & (M_1, M_2), M_3 & & ; \textit{associativity} \end{array}$$

Fig. 3. Rules of the γ_0-calculus

reduction, which applies a γ-abstraction to a solution. A molecule $(\gamma\langle x\rangle.M), \langle N\rangle$ can be reduced only if:

Inert(N): the content N of the solution argument is a closed term made exclusively of γ-abstractions or exclusively of solutions (which may be active),

or *Hidden*(x, M): the variable x occurs in M only as $\langle x\rangle$. Therefore $\langle N\rangle$ can be active since no access is done to its contents.

So, a molecule can be extracted from its enclosing solution only when it has reached an inert state. This is an important restriction that permits the ordering of rewritings. Without this restriction, the contents of a solution could be extracted in any state and the solution construct would lose its purpose. Reactions can occur in parallel as long as they apply to disjoint sub-terms. A molecule is in normal form if all its molecules are inert.

Consider, for example, the following molecules:

$$w \equiv \gamma\langle x\rangle.x, \langle x\rangle \qquad \Omega \equiv w, \langle w\rangle \qquad I \equiv \gamma\langle x\rangle.\langle x\rangle$$

Clearly, Ω is an always active (non terminating) molecule and I an inert molecule (the identity function in normal form). The molecule $\langle\Omega\rangle, \langle I\rangle, \gamma\langle x\rangle.\gamma\langle y\rangle.x$ reduces as follows:

$$\langle\Omega\rangle, \langle I\rangle, \gamma\langle x\rangle.\gamma\langle y\rangle.x \longrightarrow \langle\Omega\rangle, \gamma\langle y\rangle.I \longrightarrow I$$

The first reduction is the only one possible: the γ-abstraction extracts x from its solution and $\langle I\rangle$ is the only inert molecule ($\mathit{Inert}(I) \wedge \neg\mathit{Hidden}(x, \gamma\langle y\rangle.x)$). The second reduction is possible only because the active solution $\langle\Omega\rangle$ is not extracted but removed ($\neg\mathit{Inert}(\Omega) \wedge \mathit{Hidden}(y, I)$)

2.2 Two fundamental extensions

The γ_0-calculus is a quite expressive higher-order calculus. However, compared to the original Gamma [2] and other chemical models [5,6], it lacks two fundamental features:

- *Reaction condition.* In Gamma, reactions are guarded by a condition that must be fulfilled in order to apply them. Compared to γ_0 where inertia and termination are described syntactically, conditional reactions give these notions a semantic nature.
- *Atomic capture.* In Gamma, any fixed number of elements can take part in a reaction. Compared to a γ_0-abstraction which reacts with one element at a time, a n-ary reaction takes atomically n elements which cannot take part in any other reaction at the same time.

These two extensions are orthogonal and enhance greatly the expressivity of chemical calculi. So from now, we consider the γ-calculus extended with booleans, integers, arithmetic and booleans operators, tuples (written $x_1{:}\ldots{:}x_n$) and the possibility of naming molecules ($\mathit{ident} = M$). Furthermore, γ-abstractions (also called *active molecules*) can react according to a condition and can extract elements using pattern-matching. The syntax of γ-abstractions is extended to:

$$\gamma P\lfloor C\rfloor.M$$

where M is the action, C is the reaction condition and P a pattern extracting the elements participating in the reaction. Patterns have the following syntax:

$$P ::= x \mid \omega \mid \mathit{ident} = P \mid P, P \mid \langle P\rangle$$

where

- variables (x) match basic elements (integers, booleans, tuples, ...),
- ω is a named wild card that matches any molecule (even the empty one),
- *ident* $= P$ matches any molecule m named *ident* which matches P,
- P_1, P_2 matches any molecule (m_1, m_2) such that m_1 matches P_1 and m_2 matches P_2,
- $\langle P \rangle$ matches any solution $\langle m \rangle$ such that m matches P.

For example, the pattern $\mathrm{Sol} = \langle x, y, \omega \rangle$ matches any solution named *Sol* containing at least two basic elements named x and y. The rest of the solution (that may be empty) is matched by ω.

γ-abstractions are one-shot: they are consumed by the reaction. However, many programs are naturally expressed by applying the same reaction an arbitrary number of times. We introduce recursive (or n-shot) γ-abstractions which are not consumed by the reaction. We denote them by the following syntax:

$$\textbf{replace } P \textbf{ by } M \textbf{ if } C$$

Such a molecule reacts exactly as $\gamma P \lfloor C \rfloor . M$ except than it remains after the reaction and can be used as many times as necessary. If needed, they can be removed by another molecule, thanks to the higher-order nature of the language. If the condition C is **true**, we omit it in the definition of one-shot or n-shot molecules.

A higher-order Gamma program is an unstable solution of molecules. The execution of that program consists in performing the reactions (modulo A/C) until a stable state is reached (no more reaction can occur).

3 Towards an autonomic mail system

In this section, we describe an autonomic mail system within the Gamma framework. This example illustrate the adequacy of the chemical paradigm to the description of autonomic systems.

3.1 General description: self-organization.

The mail system consists in servers, each one dealing with a particular address domain, and clients sending their messages to their domain server. Servers forward messages addressed to other domains to the network. They also get messages addressed to their domain from the network and direct them to the appropriate clients. The mail system (see Figure 4) is described using several molecules:

- Messages exchanged between clients are represented by basic molecules whose structure is left unspecified. We just assume that relevant information (such as sender's address, recipient's address, etc.) can be extracted using appropriate functions (such as *sender*, *recipient*, *senderDomain*, etc.).
- Solutions named ToSend_{d_i} contain the messages to be sent by the client i of domain d.

- Solutions named Mbox_{d_i} contain the messages received by the client i of domain d.
- Solutions named Pool_d contain the messages that the server of domain d must take care of.
- The solution named Network represents the global network interconnecting domains.
- A client i in domain d is represented by two active molecules send_{d_i} and recv_{d_i}.
- A server of a domain d is represented by two active molecules put_d and get_d.

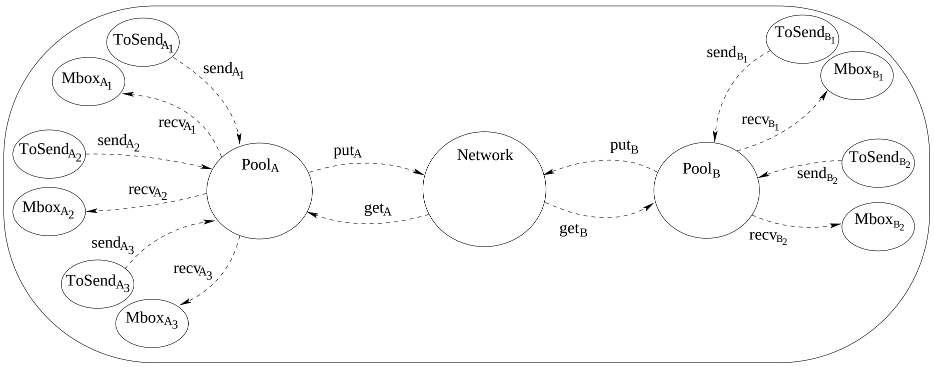


Fig. 4. Mail system.

Clients send messages by adding them to the pool of messages of their domain. They receive messages from the pool of their domain and store them in their mailbox. The send_{d_i} molecule sends messages of the client i (*i.e.,* messages in the ToSend_{d_i} solution) to the client's domain pool (*i.e.,* the Pool_d solution). The recv_{d_i} molecule places the messages addressed to client i (*i.e.,* messages in the Pool_d solution whose recipient is i) in the client's mailbox (*i.e.,* the Mbox_{d_i} solution).

Servers forward messages from their pool to the network. They receive messages from the network and store them in their pool. The put_d molecule forwards only messages addressed to other domains than d. The molecule get_d extracts messages addressed to d from the network and places them in the pool of domain d. The system is a solution, named MailSystem, containing molecules representing clients, messages, pools, servers, mailboxes and the network. Figure 4 represents graphically the solution with five clients grouped into two domains A and B and Figure 5 provides the definition of the molecules.

$$
\begin{aligned}
\mathrm{send}_{d_i} &= \textbf{replace}\ \mathrm{ToSend}_{d_i} = \langle msg, \omega_t \rangle,\ \mathrm{Pool}_d = \langle \omega_p \rangle \\
&\quad\ \textbf{by}\ \mathrm{ToSend}_{d_i} = \langle \omega_t \rangle,\ \mathrm{Pool}_d = \langle msg, \omega_p \rangle \\
\mathrm{recv}_{d_i} &= \textbf{replace}\ \mathrm{Pool}_d = \langle msg, \omega_p \rangle, \mathrm{Mbox}_{d_i} = \langle \omega_b \rangle \\
&\quad\ \textbf{by}\ \mathrm{Pool}_d = \langle \omega_p \rangle, \mathrm{Mbox}_{d_i} = \langle msg, \omega_b \rangle \\
&\quad\ \textbf{if}\ recipient(msg) = i \\
\mathrm{put}_d &= \textbf{replace}\ \mathrm{Pool}_d = \langle msg, \omega_p \rangle, \mathrm{Network} = \langle \omega_n \rangle \\
&\quad\ \textbf{by}\ \mathrm{Pool}_d = \langle \omega_p \rangle, \mathrm{Network} = \langle msg, \omega_n \rangle \\
&\quad\ \textbf{if}\ recipientDomain(msg) \neq d \\
\mathrm{get}_d &= \textbf{replace}\ \mathrm{Network} = \langle msg, \omega_n \rangle, \mathrm{Pool}_d = \langle \omega_p \rangle \\
&\quad\ \textbf{by}\ \mathrm{Network} = \langle \omega_n \rangle, \mathrm{Pool}_d = \langle msg, \omega_p \rangle \\
&\quad\ \textbf{if}\ recipientDomain(msg) = d
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{MailSystem} = \langle\ & \mathrm{send}_{A_1},\ \mathrm{recv}_{A_1},\ \mathrm{ToSend}_{A_1} = \langle \ldots \rangle,\ \mathrm{Mbox}_{A_1} = \langle \ldots \rangle, \\
& \mathrm{send}_{A_2},\ \mathrm{recv}_{A_2},\ \mathrm{ToSend}_{A_2} = \langle \ldots \rangle,\ \mathrm{Mbox}_{A_2} = \langle \ldots \rangle, \\
& \mathrm{send}_{A_3},\ \mathrm{recv}_{A_3},\ \mathrm{ToSend}_{A_3} = \langle \ldots \rangle,\ \mathrm{Mbox}_{A_3} = \langle \ldots \rangle, \\
& \mathrm{put}_A,\ \mathrm{get}_A,\ \mathrm{Pool}_A,\ \ \mathrm{Network},\ \ \mathrm{put}_B,\ \mathrm{get}_B,\ \mathrm{Pool}_B, \\
& \mathrm{send}_{B_1},\ \mathrm{recv}_{B_1},\ \mathrm{ToSend}_{B_1} = \langle \ldots \rangle,\ \mathrm{Mbox}_{B_1} = \langle \ldots \rangle, \\
& \mathrm{send}_{B_2},\ \mathrm{recv}_{B_2},\ \mathrm{ToSend}_{B_2} = \langle \ldots \rangle,\ \mathrm{Mbox}_{B_2} = \langle \ldots \rangle \\
\rangle &
\end{aligned}
$$

Fig. 5. Self-organization molecules.

3.2 Self-healing.

We now assume that a server may crash. To prevent the mail service from being discontinued, we add an emergency server for each domain (see Figure 6). The

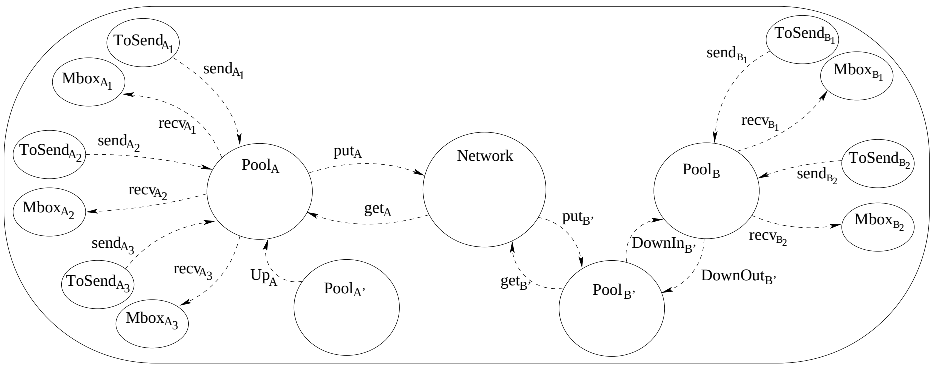


Fig. 6. Highly-available mail system.

$$\begin{aligned}
\text{crashServer}_d = {} & \textbf{replace}\ \text{put}_d,\ \text{get}_d,\ \text{Up}_d\\
& \textbf{by}\ \text{put}_{d'},\ \text{get}_{d'}, \text{DownIn}_d,\ \text{DownOut}_d\\
& \textbf{if}\ \mathit{failure}(d)\\
\text{repairServer}_d = {} & \textbf{replace}\ \text{put}_{d'},\ \text{get}_{d'}, \text{DownIn}_d,\ \text{DownOut}_d\\
& \textbf{by}\ \text{put}_d,\ \text{get}_d,\ \text{Up}_d\\
& \textbf{if}\ \mathit{recover}(d)\\
\text{DownOut}_d = {} & \textbf{replace}\ \text{Pool}_d = \langle msg, \omega_p\rangle, \text{Pool}_{d'} = \langle \omega_n\rangle\\
& \textbf{by}\ \text{Pool}_d = \langle \omega_p\rangle, \text{Pool}_{d'} = \langle msg, \omega_n\rangle\\
& \textbf{if}\ \mathit{domain}(msg) \neq d\\
\text{DownIn}_d = {} & \textbf{replace}\ \text{Pool}_d = \langle \omega_p\rangle, \text{Pool}_{d'} = \langle msg, \omega_n\rangle\\
& \textbf{by}\ \text{Pool}_d = \langle msg, \omega_p\rangle, \text{Pool}_{d'} = \langle \omega_n\rangle\\
& \textbf{if}\ \mathit{domain}(msg) = d\\
\text{Up}_d = {} & \textbf{replace}\ \text{Pool}_{d'} = \langle msg, \omega_p\rangle, \text{Pool}_d = \langle \omega_n\rangle\\
& \textbf{by}\ \text{Pool}_{d'} = \langle \omega_p\rangle, \text{Pool}_d = \langle msg, \omega_n\rangle
\end{aligned}$$

$$\begin{aligned}
\text{MailSystem} = \langle \ldots,\ \text{Up}_\text{A},\ \text{Up}_\text{B},\ \text{Pool}'_\text{A},\ \text{Pool}'_\text{B}, & \ \text{crashServer}_\text{A},\ \text{repairServer}_\text{A},\\
& \ \text{crashServer}_\text{B},\ \text{repairServer}_\text{B}\rangle
\end{aligned}$$

Fig. 7. Self-healing molecules.

emergency servers work with their own pool as usual but are active only when the corresponding main server has crashed. The modeling of a server crash can be done using the reactive molecules described in Figure 7. When a failure occurs, the active molecules representing a main server are replaced by molecules representing the corresponding emergency server. The boolean *failure* denotes a (potentially complex) failure detection mechanism. The inverse reaction repairServer represents the recovery of the server.

The two molecules Up_d and $(\text{DownIn}_d, \text{DownOut}_d)$ represent the state of the main server d in the solution, but they are also active molecules in charge of transferring pending messages from Pool_d to $\text{Pool}_{d'}$; then, they may be forwarded by the emergency server.

The molecule DownOut_d transfers all messages bound to another domain than d from the main pool Pool_d to the emergency pool $\text{Pool}_{d'}$. The molecule DownIn_d transfers all messages bound to the domain d from the emergency pool $\text{Pool}_{d'}$ to the main pool Pool_d.

After a transition from the *Down* state to the *Up* state, it may remain some messages in the emergency pools. So, the molecule Up_d brings back all the messages of the emergency pool $\text{Pool}_{d'}$ into the the main pool Pool_d to be then treated by the repaired main server. In our example, self-healing can be implemented by two emergency servers A' and B' and boils down to adding the molecules of Figure 7 into the main solution.

Other self-management features have been developed in [7]: self-optimization (by enabling the emergency server and load-balancing messages between it and the main server), self-protection (detect and suppress spams) and self-configuration (managing mobile clients).

Our description should be regarded as a high-level parallel and modular specification. It allows to design and reason about autonomic systems at an appropriate level of abstraction. Let us emphasize the beauty of the resulting programs which rely essentially on the higher-order and chemical nature of Gamma. A direct implementation of this program is likely to be quite inefficient and further refinements are needed; this is another exciting research direction, not tackled here.

4 Conclusion

We have presented a higher-order multiset transformation language which can be described using a chemical reaction metaphor. The higher-order property of our model makes it much more powerful and expressive than the original Gamma [2] or than the Linda language as described in [8]. In this article, we have shown the fundamental features of the chemical programming paradigm. The γ_0-calculus embodies the essential characteristics (AC multiset rewritings) in only four syntax rules. This minimal calculus has been shown to be expressive enough to express the λ-calculus and a large class of non-deterministic programs [4]. However, in order to come close to a real chemical language, two extensions must be considered: reaction conditions and atomic capture. With appropriate syntactic sugar (recursion, constants, operators, pattern-matching, etc.), the extended calculus can easily express most of the existing chemical languages.

In this higher-order model, molecules representing reaction rules can be seen as catalysts that perform computations and implements new features. This programming style has been illustrated by the example of an autonomic mail system described as molecules and transformation rules. These rules may apply as soon as a predefined condition holds without external intervention. In other words, the system configures and manages itself to face predefined situations. Our chemical mail system shows that our approach is well-suited to the high-level description of autonomic systems. Reaction rules exhibit the essence of "autonomy" without going into useless details too early in the development process.

An interesting research direction is to take advantage of these high-level descriptions to carry out proofs of properties of autonomic systems (in the same spirit as [9]). For example, "not losing any messages" would be an important property to prove for our mail system. Another direction would be to extend our language to prevent clumsy encodings (*e.g.,* using advanced data structures and others high-level facilities).

References

1. Banâtre, J.P., Le Metayer, D.: A new computational model and its discipline of programming. Technical Report RR0566, INRIA (1986)

2. Banâtre, J.P., Le Metayer, D.: Programming by multiset transformation. Communications of the ACM (CACM) **36** (1993) 98–111
3. Banâtre, J.P., Fradet, P., Le Metayer, D.: Gamma and the chemical reaction model: Fifteen years after. In: Multiset Processing. Volume 2235 of LNCS., Springer-Verlag (2001) 17–44
4. Banâtre, J.P., Fradet, P., Radenac, Y.: Principles of chemical programming. In: RULE'04. ENTCS, Elsevier (2004) (to appear).
5. Le Metayer, D.: Higher-order multiset programming. In (AMS), A.M.S., ed.: Proc. of the DIMACS workshop on specifications of parallel algorithms. Volume 18 of Dimacs Series in Discrete Mathematics. (1994)
6. Păun, G.: Computing with membranes. Journal of Computer and System Sciences **61** (2000) 108–143
7. Banâtre, J.P., Fradet, P., Radenac, Y.: Chemical specification of autonomic systems. In: IASSE'04. (2004) (to appear).
8. Carriero, N., Gelernter, D.: Linda in Context. Communications of the ACM **32** (1989) 444–458
9. Barradas, H.: Systematic derivation of an operating system kernel in Gamma. Phd thesis (in french), University of Rennes, France (1993)

Enabling Autonomic Applications: Models and Infrastructure [*]

Manish Parashar, Zhen Li, Hua Liu, Cristina Schmidt, Vincent Matossian and Nanyan Jiang

The Applied Software Systems Laboratory
Rutgers University, Piscataway NJ 08904, USA,
parashar@caip.rutgers.edu

Abstract. The increasing complexity, heterogeneity and dynamism of networks, systems and applications have made our computational and information infrastructure brittle, unmanageable and insecure. This has necessitated the investigation of a new paradigm for system and application design, which is based on strategies used by biological systems to deal with similar challenges of complexity, heterogeneity, and uncertainty, i.e. autonomic computing. This paper introduces Project AutoMate. The overall goal of Project AutoMate is to enable the development and execution of self-managed autonomic Grid applications. It supports the definition of autonomic elements, the development of autonomic applications as the dynamic and opportunistic composition of these autonomic elements, and the policy, content and context driven definition, execution and management of these applications. In this paper we introduce the key components of AutoMate and describe their underlying conceptual models and implementations.

1 Introduction

The emergence of pervasive wide-area distributed computing environments, such as pervasive information systems and computational Grid, has enabled a new generation of applications that are based on seamless access, aggregation and interaction. For example, it is possible to conceive a new generation of scientific and engineering simulations of complex physical phenomena that symbiotically and opportunistically combine computations, experiments, observations, and real-time data, and can provide important insights into complex systems such as interacting black holes and neutron stars, formations of galaxies, and subsurface flows in oil reservoirs and aquifers, etc. Other examples include pervasive applications that leverage the pervasive information Grid to continuously manage,

[*] The research presented in this paper is supported in part by the National Science Foundation via grants numbers ACI 9984357 (CAREERS), EIA 0103674 (NGS), EIA-0120934 (ITR), ANI-0335244 (NRT), CNS-0305495 (NGS) and by DOE ASCI/ASAP (Caltech) via grant number 82-1052856.

adapt, and optimize our living context, crisis management applications that use pervasive conventional and unconventional information for crisis prevention and response, medical applications that use in-vivo and in-vitro sensors and actuators for patient management, and business applications that use anytime-anywhere information access to optimize profits.

However, the underlying computing environment is inherently large, complex, heterogeneous and dynamic, globally aggregating large numbers of independent computing and communication resources, data stores and sensor networks. Furthermore, these emerging applications are similarly complex and highly dynamic in their behaviors and interactions. Together, these characteristics result in application development, configuration and management complexities that break current paradigms based on passive components and static compositions. Clearly, there is a need for a fundamental change in how these applications are developed and managed. This has led researchers to consider alternative programming paradigms and management techniques that are based on strategies used by biological systems to deal with complexity, dynamism, heterogeneity and uncertainty. The approach, referred to as autonomic computing, aims at realizing computing systems and applications capable of managing themselves with minimum human intervention.

The overall goal of Project AutoMate is to investigate conceptual models and implementation architectures that can enable the development and execution of such self-managed autonomic applications. Specifically, it investigates programming models and frameworks and middleware services that support the definition of autonomic elements, the development of autonomic applications as the dynamic and opportunistic composition of these autonomic elements, and the policy, content and context driven definition, execution and management of these applications.

A schematic overview of AutoMate [1] is presented in Figure 1. In this paper, we introduce AutoMate and its key components, and describe their underlying conceptual models and implementations. Specfically, we describe the Accord programming framework, the Rudder decentralized coordination framework and deductive engine, and the Meteor content-based middleware providing support for content-based routing, discovery and associative messaging. Project AutoMate additionally includes the Sesame [6] context-based security infrastructure, the DIAS self-protection service and the Discover/DIOS collaboratory services [8], which are not discussed in this paper.

2 Accord, A Programming Framework for Autonomic Applications

Accord programming framework [3] consists of four concepts: (1) an application context that defines a common semantic basis for components and the application, (2) definition of autonomic components as the basic building blocks for an autonomic application, (3) definition of rules and mechanisms for the management and dynamic composition of autonomic components, and (4) rule enforcement to enable autonomic application behaviors.

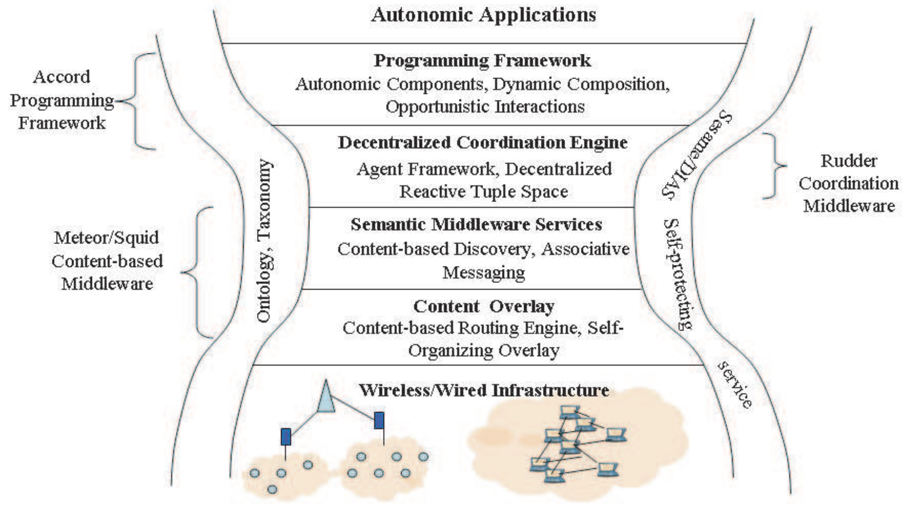

Autonomic Applications
Programming Framework
Autonomic Components, Dynamic Composition, Opportunistic Interactions
Decentralized Coordination Engine
Agent Framework, Decentralized Reactive Tuple Space
Semantic Middleware Services
Content-based Discovery, Associative Messaging
Content Overlay
Content-based Routing Engine, Self-Organizing Overlay
Wireless/Wired Infrastructure
Accord Programming Framework
Meteor/Squid Content-based Middleware
Rudder Coordination Middleware
Ontology, Taxonomy
Sesame/DIAS Self-protecting service

Fig. 1. AutoMate Architecture

Application Context: Autonomic components should agree on common semantics for defining and describing application namespaces, and component interfaces, sensors and actuators. Using such a common context allows definition of rules for autonomic management of components and dynamic composition and interactions between the components. In Accord, functional and non-functional aspects of components are described using an XML-based language.

Autonomic Component: An autonomic component is the fundamental building block for an autonomic application. It extends the traditional definition of components to define a self-contained modular software unit of composition with specified interfaces and explicit context dependencies. Additionally, an autonomic component encapsulates rules, constraints and mechanisms for self-management and dynamically interacts with other components and the system. An autonomic component shown in Figure 2 is defined by 3 ports:

– A *functional port* that defines the functional behaviors provided and used by the component;
– A *control port* that defines a set of sensors and actuators exported by the component for external monitoring and control, and a constraint set that defines the access to and operation of the sensors/actuators based on state, context and/or high-level access policies;
– An *operational port* that defines the interfaces to formulate, inject and manage the rules which are used to manage the component runtime behaviors.

These aspects enhance component interfaces to export information and policies about their behaviors, resource requirements, performance, interactivity and

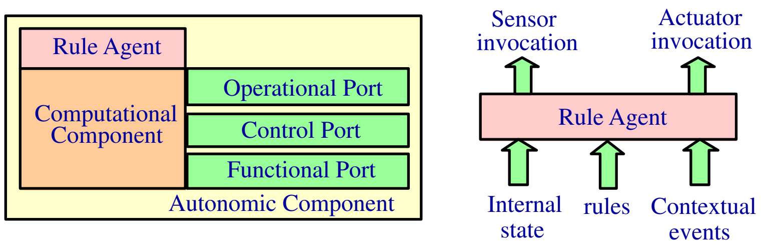


Fig. 2. An autonomic component

adaptability to system and application dynamics. An embedded rule agent monitors the component's state and controls the execution of rules. Rule agents cooperate across application compositions to fulfill overall application objectives.

Rule Definition: Rules incorporate high-level guidance and practical human knowledge in the form of an IF-THEN expression. The condition part of a rule is a logical combination of component/environment sensors and events. The action part of a rule consists of a sequence of component/system sensor/actuator invocations. A rule fires when its condition expression evaluates to be true and the corresponding actions are executed. Two class of rules are defined: (1) *Behavioral rules* that control the runtime functional behaviors of an autonomic component and (2) *Interaction rules* that control the interactions among components, between components and their environment, and the coordination within an autonomic application.

Rule Enforcement: Rules are injected into components at run time and enable self-managing behavior for an autonomic application. Behavioral rules are executed by a rule agent embedded within a single component without affecting other components. Interaction rules define interactions among components. For each interaction pattern, a set of interaction rules are defined and dynamically injected into the interacting components. The coordinated execution of these rules result in the realization of interaction and coordination behaviors between the components.

The key concepts of Accord have been prototyped and evaluated in the context of distributed scientific/engineering simulations as the part of the DIOS++ project [2]. In this prototype, computational objects were enhanced with sensors, actuators and behavior rules. The objects can be dynamically created, deleted or migrated, and rules can span multiple objects across multiple processors.

3 Rudder, An Agent-based Coordination Middleware

Rudder [4] is an agent-based middleware infrastructure for autonomic Grid applications. The goal of Rudder is to provide the core capabilities for supporting autonomic compositions, adaptations, and optimizations. Specifically, Rudder employs context-aware software agents and a decentralized tuple space coordination model to enable context and self awareness, application monitoring and analysis, and policy definition and its distributed execution. Rudder effectively supports the Accord programming framework and enables self-managing autonomic applications. The overall architecture builds on two concepts:

– Context-aware agents provide context information at different system and application levels to trigger autonomic behaviors. Agents can control, compose and manage autonomic components, monitor and analyze system runtime state, sense changes in environment and application requirements, dynamically define and enforce rules to locally enable component self-managing behaviors.
– A robust decentralized reactive tuple space can scalably and reliably support distributed agent coordination. It provides mechanisms for deploying and routing rules, decomposing and distributing them to relevant agents, and enforcing the interactions between components that are required for global system and application properties to emerge.

Agent framework: The Rudder agent framework consists of three types of peer agents: Component Agent (CA), System Agent (SA), and Composition Agent (CSA). CA and SA exist as system services, while composition agents are transient and are generated to satisfy specific application requirements. CAs manage the computations performed locally within components and define interaction rules to specify component interaction and communication behaviors and mechanisms. They are integrated with component rule agents to provide components with uniform access to middleware services, control their functional and interaction behaviors and manage their life cycles. SAs are embedded within Grid resource units, exist at different levels of the system, and represent their collective behaviors. CSAs enable dynamic composition of autonomic components by defining and executing workflow-selection and component-selection rules. Workflow-selection rules are used to select appropriate composition plans to enact, which are then inserted as reactive tuples into the tuple space. Component-selection rules are used to semantically discover, and select registered components, allowing tasks to optimize the execution and tolerate some failure in components, connections, and hosts. CSAs negotiate to decide interaction patterns for a specific application workflow and coordinate with the associated component agents to execute the interaction rules at runtime. This enables autonomic applications to dynamically change flows, components and component interactions to address application and system dynamics and uncertainty.

Decentralized tuple space: The Rudder decentralized reactive tuple space (DRTS) provides the coordination service for distributed agents, and mechanisms for rule definition, deployment and enforcement. Rudder extends the traditional tuple space with a distributed, guaranteed and flexible content-based matching engine and reactive semantics to enable global coordination in dynamic and ad hoc agent communities. Runtime adaptive polices defined by the context-aware agents can be inserted and executed using reactive tuples to achieve coordinated application execution and optimized computational resource allocation and utilization. The DRTS builds on a resilient self-organizing peer-to-peer content-based overlay, and supports the definition and execution of coordination policies through programmable reactive behaviors. These behaviors are dynamically defined using stateful reactive tuples. A reactive tuple consists of three parts: (1) *Condition* associates reactions to triggering events, (2) *Reaction* specifies the computation associated with the tuple's reactive behavior, and (3) *Guard* defines the execution semantics of the reactive behavior (e.g., immediately and once). Policies and constraints dynamically defined by administrators or agents can be triggered and executed to satisfy specific application requirements (e.g., prevent undesired malicious operations to defend system integrity). The current prototype of the DRTS builds on Meteor infrastructure decribed below.

4 Meteor: A Content-based Middleware

Meteor [5] is a scalable content-based middleware infrastructure that provides services for content routing, content discovery and associative interactions. A schematic overview of the Meteor stack is presented in Figure 3. It consists of 3 key components: (1) a self-organizing overlay, (2) a content-based routing and discovery infrastructure (Squid), and (3) the Associative Rendezvous Messaging Substrate (ARMS). The Meteor overlay network is composed of Rendezvous

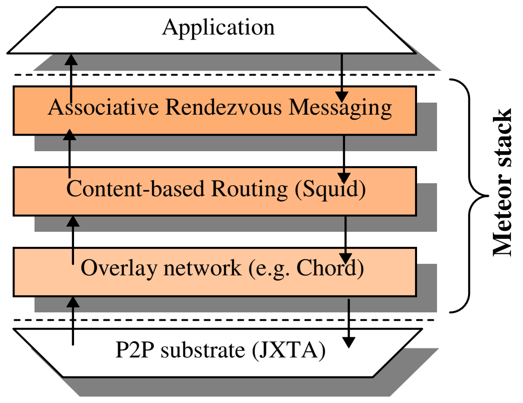

Application
Associative Rendezvous Messaging
Content-based Routing (Squid)
Overlay network (e.g. Chord)
P2P substrate (JXTA)
Meteor stack

Fig. 3. A schematic overview of the Meteor stack

Peer (RP) nodes, which may be access points or message forwarding nodes in ad-hoc sensor networks and servers or end-user computers in wired networks. RP nodes can join or leave the network at any time. The current Meteor overlay network uses Chord to provide a single operation: **lookup**(*identifier*) which requires an exact identifier from the layers above. Given an identifier, this operation locates the node that is responsible for it (e.g., the node with an identifier that is the closest identifier greater than or equal to the queried identifier). However, note that the upper layers of Meteor can work with different overlay structures.

Squid [7] is the Meteor content-based routing engine and decentralized information discovery service. It support flexible content-based routing and complex queries containing partial keywords, wildcards, and ranges, and guarantees that all existing data elements that match a query will be found with bounded costs in terms of number of messages and number of nodes involved. The key innovation of Squid is the use of a locality preserving and dimension reducing indexing scheme, based on the Hilbert Space Filling Curve (SFC), which effectively maps the multidimensional information space to the peer identifier space. Squid effectively maps complex queries consisting of keyword tuples (multiple keywords, partial keywords, wildcards, and ranges) onto clusters of identifiers, and guarantees that all peers responsible for identifiers in these clusters will be found with bounded costs in terms of number of messages and the number of intermediate RP nodes involved. Keywords can be common words or values of globally defined attributes, depending on the nature of the application that uses Squid, and are based on common ontologies and taxonomies.

The ARMS layer implements the Associative Rendezvous (AR) interaction paradigm. AR is a paradigm for content-based decoupled interactions with programmable reactive behaviors. Rendezvous-based interactions provide a mechanism for decoupling senders and receivers. Senders send messages to a rendezvous point without knowledge of which or where the receivers are. Similarly, receivers receive messages from a rendezvous point without knowledge of which or where the senders are. Note that senders and receivers may be decoupled in both space and time. Such decoupled asynchronous interactions are naturally suited for large, distributed, and highly dynamic systems such as pervasive Grid environments.

AR extends the conventional name/identifier-based rendezvous in two ways. First, it uses flexible combinations of keywords (i.e, keyword, partial keyword, wildcards, ranges) from a semantic information space, instead of opaque identifiers (names, addresses) that have to be globally known. Interactions are based on content described by keywords, such as the type of data a sensor produces (temperature or humidity) and/or its location, the type of functionality a service provides and/or its QoS guarantees, and the capability and/or the cost of a resource. Second, it enables the reactive behaviors at the rendezvous points to be encapsulated within messages increasing flexibility and expressibility, and enabling multiple interaction semantics (e.g. broadcast, multicast, notification, publisher/subscriber, mobility, etc.).

At each Meteor RP, ARMS consists of two components: the *profile manager* and the *matching engine*. Meteor messaging layer receives the application specific profiles embedded as a header in the message. All communication from applications of information provider and consumer is carried out by executing the *post* primitive, which includes in the outgoing message a sender-specified profile in the header. It will also be encapsulated with *data* dependent on the application to the messaging system; when the matched message is available from the messaging system, it will be sent to the application for further processing. For example, at the Associative Rendezvous Point, *action(notify)* will be executed when information becomes available.

The current implementation of Meteor builds on Project JXTA and has been deployed on a distributed testbed with about 100 nodes. Initial evaluations of Meteor using simulations and the testbed demonstrate its scalability and effectiveness as a paradigm for pervasive Grid environments.

5 Conclusion

In this paper, we introduced Project AutoMate and described its key components. The overarching goal of AutoMate is to enable the development, deployment and management of autonomic self-managing applications in widely distributed and highly dynamic Grid computing environments. Specifically, AutoMate provides conceptual and implementation models for the definition of autonomic elements, the development of autonomic applications as the dynamic and opportunistic composition of these autonomic elements, and the policy, content and context driven definition, execution and management of these applications.

The core components of AutoMate have been prototyped and are being currently used to enable self-managing applications in science and engineering (e.g. autonomic oil reservoir optimizations, autonomic runtime management of adaptive simulations, etc.) and to enable sensor-based pervasive applications. Further information about AutoMate and its components can be obtained from http://automate.rutgers.edu/.

References

1. Agarwal, M. and et all.: AutoMate: Enabling Autonomic Applications on the Grid. *In Proceedings of the 5th Annual International Active Middleware Services Workshop*, Seattle, WA, (2003).
2. Liu, H. and Parashar, M.: DIOS++: A Framework for Rule-Based Autonomic Management of Distributed Scientific Applications. *International Conference on Parallel and Distributed Computing*,Klagenfurt, Austria, (2003).
3. Liu, H. and Parashar, M.: A Component Based Programming Framework for Autonomic Applications. *In Proceedings of the International Conference on Autonomic Computing*, New York, NY, (2004).
4. Li, Z. and Parashar, M.: Rudder: A Rule-based Multi-agent Infrastructure for Supporting Autonomic Grid Applications. *In Proceedings of the International Conference on Autonomic Computing*, New York, NY, (2004).
5. Jiang, N., Schmidt, C., Matossian, V. and Parashar, M.:Content-based Middleware for Decoupled Interactions in Pervasive Envionments, *Technical Report Number-252*, Rutgers University, Wireless Information Network Laboratory (WINLAB),(2004).
6. Zhang, G. and Parashar, M.: Dynamic Context-aware Access Control for Grid Applications. *In Proceedings of the 4th International Workshop on Grid Computing (Grid 2003), IEEE Computer Society Press*, Phoenix, AZ, (2003).
7. Schmidt, C. and Parashar, M.:Flexible Information Discovery in Decentralized Distributed Systems, *In Proceedings of the 12th High Performance Distributed Computing (HPDC)*, Seattle, WA, (2003).
8. Bhat, V. and Parashar, M.:A Middleware Substrate for Integrating Services on the Grid, *Journal of Supercomputing, Special Issue on Infrastructures and Applications for Cluster and Grid Computing Environments*,(2004).

Grassroots Approach to Self-Management in Large-Scale Distributed Systems[*]

Ozalp Babaoglu, Márk Jelasity[**], and Alberto Montresor

Department of Computer Science, University of Bologna, Italy
e-mail: babaoglu,jelasity,montreso@cs.unibo.it

Abstract. Traditionally, autonomic computing is envisioned as replacing the human factor in the deployment, administration and maintenance of computer systems that are ever more complex. Partly to ensure a smooth transition, the design philosophy of autonomic computing systems remains essentially the same as traditional ones, only autonomic components are added to implement functions such as monitoring, error detection, repair, etc. In this position paper we outline an alternative approach which we call "grassroots self-management". While this approach is by no means a solution to all problems, we argue that recent results from fields such as agent-based computing, the theory of complex systems and complex networks can be efficiently applied to achieve important autonomic computing goals, especially in very large and dynamic environments. Unlike in traditional compositional design, the desired properties like self-healing and self-optimization are not programmed explicitly but rather "emerge" from the local interactions among the system components. Such solutions are potentially more robust to failures, are more scalable and are extremely simple to implement. We discuss the practicality of grassroots autonomic computing through the examples of data aggregation, topology management and load balancing in large dynamic networks.

1 Introduction

The desire to build fault tolerant computer systems with an intuitive and effi cient user interface has always been part of the research agenda of computer science. Still, the current scale and heterogeneity of computer systems is becoming alarming, especially because our everyday life has come to depend on such systems to an increasing degree. There is a general feeling in the research community that to cope with this new situation—which emerged as a result of Moore's Law, the widespread adoption of the Internet and computing becoming pervasive in general—needs radically new approaches to achieve seamless and effi cient functioning of computer systems.

Accordingly, more and more effort is devoted to tackle the problem of self-management. One of the most influential and widely publicized approach is IBM's *autonomic computing* initiative, launched in 2001 [8]. The term "autonomic" is a biological analogy referring to the autonomic nervous system. The function of this system in our body

[*] This work was partially supported by the Future & Emerging Technologies unit of the European Commission through Projects BISON (IST-2001-38923) and DELIS (IST-001907).

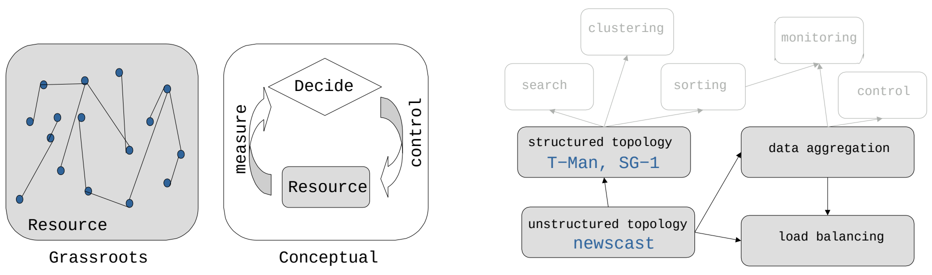
[**] also with RGAI, MTA SZTE, Szeged, Hungary

is to control “routine” tasks like blood pressure, hormone levels, heart rate, breathing rate, etc. At the same time, our conscious mind can focus on high level tasks like planning and problem solving. The idea is that autonomic computing should do just the same: computer systems would take care of routine tasks themselves while system administrators and users would focus on the actual task instead of spending most of their time troubleshooting and tweaking their systems.

Since the original initiative, the term has been adopted by the wider research community although it is still strongly associated with IBM and, more importantly, IBM’s specifi c approach to autonomic computing. It is somewhat unfortunate because the term *autonomic* would allow for a much deeper and more far-reaching interpretation, as we explain soon. In short, we should not only take it seriously *what* the autonomic nervous system does but also *how* it does it. We believe that the remarkably successful self-management of the autonomic nervous system, and biological organisms in general lies exactly in the *way* they achieve this functionality. Ignoring the exact mechanisms and stopping at the shallow analogy at the level of function description misses some important possibilities and lessons that can be learned by computer science.

The meaning of “self”. The traditional approach to autonomic computing is to replace human system administrators with software or hardware components that continuously monitor some subsystem assigned to them, forming so called *control loops* [8] which involve monitoring, knowledge based planning and execution (see Figure 1(a)). Biological systems however achieve self-management and control through entirely different, often fully distributed and *emergent* ways of processing information. In other words, the usual biological interpretation of self-management involves no managing and managed entities. There is often no subsystem responsible for self-healing or self-optimization; instead, these properties simply follow from some simple local behavior of the components typically in a highly non-trivial way. The term “self” is meant truly in a grassroots sense, and we believe that this fact might well be the reason of many desirable properties like extreme robustness and adaptivity, with the additional benefi t of a typically very simple implementation.

Trust. There are a few practical obstacles in the way of the deployment of grassroots self-management. One is the entirely different and somewhat un-natural way of thinking and the relative lack of understanding of the principles of self-organization and emergence [11]. Accordingly, *trust delegation* represents a problem: psychologically it is more relaxing to have a single point of control, an explicit controlling entity. In the case of the autonomic nervous system we cannot do anything else but trust it, although probably many people would prefer to have more control, especially when things go wrong. Indeed, the tendency in engineering is to try to isolate and create central units that are responsible for a function. A good example is the car industry that gradually places more and more computers into our cars that explictly control the different functions, thereby replacing old and proven mechanisms that were based on some, in a sense, self-optimizing mechanism (like the carburetor) and so also sacrifi cing the self-healing and robustness features of these functions to some degree.



(a) models of self-management

(b) dependence relations between components

Fig. 1. Conceptual models of self-management and composition.

Modularity. To exploit the power and simplicity of emergent behavior yet to ensure that these mechanisms can be trusted and be incorporated in systems in an informed manner, we believe that a *modular paradigm* is required. The idea is to identify a collection of simple and predictable services as *building blocks* and combine them in arbitrarily complex functions and protocols. Such a modular approach presents several attractive features. Developers will be allowed to plug different components implementing a desired function into existing or new applications, being certain that the function will be performed in a predictable and dependable manner. Research may be focused on the development of simple and well-understood building blocks, with a particular emphasis on important properties like robustness, scalability, self-organization and self-management.

The goal of this position paper is to promote this idea by describing our preliminary experiences in this direction. Our recent work has resulted in a collection of simple and robust building blocks, which include *data aggregation* [6, 10], *membership management* [5], *topology construction* [4, 9] and *load balancing* [7]. Our building blocks are typically no more complicated than a cellular automaton or a swarm model which makes them ideal objects for research. Practical applications based on them can also benefi t from a potentially more stable foundation and predictability, a key concern in fully distributed systems. Most importantly, they are naturally self-managing, without dedicated system components. In the rest of the paper, we briefly describe these components.

2 A Collection of Building Blocks

Under the auspices of the BISON project [1], our recent activity has been focused on the identifi cation and development of protocols for several simple basic functions. The components produced so far can be informally subdivided into two broad categories: *overlay protocols* and *functional protocols*. An overlay protocol is aimed at maintaining

```
do forever
  wait(T time units)
  $p \leftarrow$ GETPEER()
  send $s$ to $p$
  $s_p \leftarrow$ receive($p$)
  $s \leftarrow$ UPDATE($s, s_p$)
```

(a) active thread

```
do forever
  $s_p \leftarrow$ receive(*)
  send $s$ to sender($s_p$)
  $s \leftarrow$ UPDATE($s, s_p$)
```

(b) passive thread

Fig. 2. The skeleton of a gossip-based protocol. Notation: s is the local state, s_p is the state of the peer p.

application-layer, connected communication topologies over a set of distributed nodes. These topologies may constitute the basis for functional protocols, whose task is to compute a specifi c function over the data maintained at nodes.

Our current bag of protocols includes: (i) protocols for organizing and managing structured topologies like super-peer based networks (SG-1 [9], grids and tori (T-MAN [4]); (ii) protocols for building unstructured networks based on the random topology (NEWSCAST [5]); (iii) protocols for the computation of a large set of aggregate functions, including maximum and minimum, average, sum, product, geometric mean, variance, etc [6, 10]; and (iv) a load balancing protocol [7].

The relationships between overlay and functional protocols may assume several different forms. Topologies may be explicitly designed to optimize the performance of a specifi c functional protocol (this is the case of NEWSCAST [5] used to maintain a random topology for aggregation protocols). Or, a functional protocol may be needed to implement a specifi c overlay protocol (in superpeer networks, aggregation can be used to identify the set of superpeers).

All the protocols we have developed so far are based on the gossip-based paradigm [2, 3]. Gossip-style protocols are attractive since they are extremely robust to both computation and communication failures. They are also extremely responsive and can adapt rapidly to changes in the underlying communication structure, just by their nature, without extra measures.

The skeleton of a generic gossip-based protocol is shown in Figure 2. Each node possesses a local state and executes two different threads. The *active* one periodically initiates an *information exchange* with a peer node selected randomly, by sending a message containing the local state and waiting for a response from the selected node. The *passive* one waits for messages sent by an initiator and replies with its local state.

Method UPDATE builds a new local state based on the previous local state and the state received during the information exchange. The output of UPDATE depends on the specifi c function implemented by the protocol. The local states at the two peers after an information exchange are not necessarily the same, since UPDATE may be non-deterministic or may produce different outputs depending on which node is the initiator.

Even though our system is not synchronous, it is convenient to talk about *cycles* of the protocol, which are simply consecutive wall clock intervals during which every node has its chance of performing an actively initiated information exchange.

In the following we describe the components. Figure 1(b) illustrates the dependence relations between them as will be described in the text as well.

2.1 Newscast

In NEWSCAST [5], the state of a node is given by a *partial view*, which is a set of peer descriptors with a fixed size c. A *peer descriptor* contains the address of the peer, along with a *timestamp* corresponding to the time when the descriptor was created.

Method GETPEER returns an address selected randomly among those in the current partial view. Method UPDATE merges the partial views of the two nodes involved in an exchange and keeps the c freshest descriptors, thereby creating a new partial view. New information enters the system when a node sends its partial view to a peer. In this step, the node always inserts its own, newly created descriptor into the partial view. Old information is gradually and automatically removed from the system and gets replaced by new information. This feature allows the protocol to "repair" the overlay topology by forgetting dead links, which by definition do not get updated because their owner is no longer active.

In NEWSCAST, the overlay topology is defined by the content of partial views. We have shown in [5] that the resulting topology has a very low diameter and is very close to a random graph with out-degree c. According to our experimental results, choosing $c = 20$ is already sufficient for very stable and robust connectivity.

We have also shown that, within a single cycle, the number of exchanges per node can be modeled by a random variable with the distribution $1 + \text{Poisson}(1)$. The implication of this property is that no node is more important (or overloaded) than others.

2.2 T-Man

Another component is T-MAN [4], a protocol for creating a large set of topologies. The idea behind the protocol is very similar to that of NEWSCAST. The difference is that instead of using the creation date (freshness) of descriptors, T-MAN applies a ranking function that ranks any set of nodes according to increasing distance from a base node. Method GETPEER returns neighbors with a bias towards closer ones, and, similarly, UPDATE keeps peers that are closer, according to the ranking.

Figure 3 illustrates the protocol, as it constructs a torus topology. In [4] it was shown that the protocol converges in logarithmic time also for network sizes as large as 2^{20} and for other topologies as well including the ring and binary tree topologies. With the appropriate ranking function T-MAN can be also applied to sort a set of numbers.

This component, T-MAN, relies on another component for generating an initial random topology which is later evolved into the desired one. In our case this service is provided by NEWSCAST.

2.3 SG-1

SG-1 [9] is yet another component based on NEWSCAST, whose task is to self-organize a *superpeer-based* network. This special kind of topology is organized through a two-level hierarchy: nodes that are faster and/or more reliable than "normal" nodes take

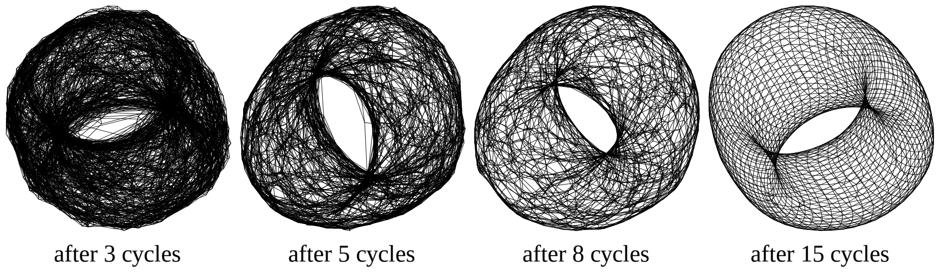


Fig. 3. Illustrative example of T-MAN constructing a torus over $50 \times 50 = 2500$ nodes, starting from a uniform random topology with $c = 20$. For clarity, only the nearest 4 neighbors (out of 20) of each node are displayed.

on server-like responsibilities and provide services to a set of clients. The superpeer paradigm allows decentralized networks to run more effi ciently by exploiting heterogeneity and distributing load to machines that can handle the burden. On the other hand, it does not inherit the flaws of the client-server model, as it allows multiple, separate points of failure, increasing the health of large-scale networks.

In our model, each node is characterized by a *capacity* parameter, that defi nes the maximum number of clients that can be served by a node. The task of SG-1 is to form a network where the role of superpeers is played by the nodes with highest capacity. All other nodes become clients of one or more superpeers. The goal is to identify the minimal set of superpeers that are able to provide the desired quality of service, based on their capacity.

In SG-1, NEWSCAST is used in two ways. First, it provides a robust underlying topology that guarantee the connectivity of the network in spite of superpeer failures. Second, NEWSCAST is used to maintain, at each node, a partial view containing a random sample of superpeers that are currently *underloaded* with respect to their capacity. At each cycle, each superpeer s tries to identify a superpeer t that (i) has more capacity than s, and (ii) is underloaded. If such superpeer exist and can be contacted, s transfers the responsibility of parts of its clients to t. If the set of clients of s ends to be empty, s becomes a client of t.

Experimental results show that this protocol converges to the target superpeer topology in logarithmic time for network sizes as large as 10^6 nodes, producing very good approximation of the target in a constant number of cycles.

2.4 Gossip-Based Aggregation

In the case of gossip-based aggregation [6, 10], the state of a node is a numeric value. In a practical setting, this value can be any attribute of the environment, such as the load or the storage capacity. The task of the protocol is to calculate an aggregate value over the set of all numbers stored at nodes. Although several aggregate functions may be computed by our protocol, in this paper provide only the details for the average function.

In order to work, this protocol needs an overlay protocol that provides an implementation of method GETPEER. Here, we assume that this service is provided by NEWSCAST, but any other overlay could be used.

To compute the average, method UPDATE(a, b) must return $(a + b)/2$. After one state exchange, the sum of the values maintained by the two nodes does not change, since they have just balanced their values. So the operation does not change the global average either; it only decreases the variance over all the estimates in the system.

In [6] it was shown that if the communication topology is not only connected but also sufficiently random, at each cycle the empirical variance computed over the set of values maintained by nodes is reduced by a factor whose expected value is $2\sqrt{e}$. Most importantly, this result is independent from the size of the network, showing the extreme scalability of the protocol.

In addition to being fast, our aggregation protocol is also very robust. Node failures may perturb the final result, as the values stored in crashed nodes are lost; but both analytical and empirical studies have shown that this effect is generally marginal [10]. As long as the overlay network remains connected, link failures do not modify the final value, they only slow down the aggregation process.

2.5 A Load-Balancing Protocol

The problem of load balancing is similar, to a certain extent, to the problem of aggregation. Each node has a certain amount of load and the nodes are allowed to transfer some portions of their load between themselves. The goal is to reach a state where each node has the same amount of load. To this end, nodes can make decisions for sending or receiving load based only on locally available information. Differently from aggregation, however, the amount of load that can be transfered in a given cycle is bounded: the transfer of a unit of load may be an expensive operation. In our present discussion, we use the term *quota* to identify this bound and we denote it by Q. Furthermore, we assume that the quota is the same at each node.

A simple, yet far from optimal idea for a completely decentralized algorithm could be based on the aggregation mechanism illustrated above. Periodically, each node contacts a random node among its neighbors. The loads of the two nodes are compared; if they differ, a quantity q of load units is transfered from the node with more load to the node with less load. q is clearly bounded by the quota Q and quantity of load units needed to balance the nodes.

If the network is connected, this mechanism will eventually balance the load among all nodes. Nevertheless, it fails to be optimal with respect to load transfers. The reason is simple: if the loads of two nodes are both higher than the average load, transferring load units from one to the other is useless. Instead, they should contact nodes whose load is smaller than the average, and perform the transfer with them.

Our load-balancing algorithm is based exactly on this intuition. The nodes obtain an estimate of the current average load through the aggregation protocol described above. This estimate is the target load; based on its value, a node may decide if it is overloaded, underloaded, or balanced. Overloaded nodes contact their underloaded neighbors in order to transfer their excess load and underloaded nodes contact their overloaded neighbors to perform the opposite operation. Nodes that have reached the target load stop participating in the protocol. Although this was a simplifi ed description, it is easy to see that this protocol is optimal with respect to load transfer, because each node transfers exactly the amount of load needed to reach its target load. As we show in [7], the protocol is also optimal with respect to speed under some conditions on the initial load distribution.

3 Conclusions

In this abstract, we presented examples for simple protocols that exhibit self-managing properties without any explicit management components or control loops; in other words, without increased complexity. We argued that a modular approach might be the way towards effi cient deployment of such protocols in large distributed systems. To validate our ideas, we have briefly presented gossip based protocols as possible building blocks: topology and membership management (T-MAN, SG-1 and NEWSCAST), aggregation, and load balancing.

References

1. The Bison Project. `http://www.cs.unibo.it/bison`.
2. A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database management. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing (PODC'87)*, pages 1–12, Vancouver, August 1987. ACM.
3. P. T. Eugster, R. Guerraoui, A.-M. Kermarrec, and L. Massoulié. Epidemic information dissemination in distributed systems. *IEEE Computer*, 37(5):60–67, May 2004.
4. M. Jelasity and O. Babaoglu. T-Man: Fast gossip-based construction of large-scale overlay topologies. Technical Report UBLCS-2004-7, University of Bologna, Department of Computer Science, Bologna, Italy, May 2004.
5. M. Jelasity, W. Kowalczyk, and M. van Steen. Newscast computing. Technical Report IR-CS-006, Vrije Universiteit Amsterdam, Department of Computer Science, Amsterdam, The Netherlands, November 2003.
6. M. Jelasity and A. Montresor. Epidemic-style proactive aggregation in large overlay networks. In *Proceedings of The 24th International Conference on Distributed Computing Systems (ICDCS 2004)*, pages 102–109, Tokyo, Japan, March 2004. IEEE Computer Society.
7. M. Jelasity, A. Montresor, and O. Babaoglu. A modular paradigm for building self-organizing peer-to-peer applications. In G. Di Marzo Serugendo, A. Karageorgos, O. F. Rana, and F. Zambonelli, editors, *Engineering Self-Organising Systems*, number 2977 in Lecture Notes in Artificial Intelligence, pages 265–282. Springer, 2004.
8. J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, January 2003.
9. A. Montresor. A robust protocol for building superpeer overlay topologies. In *Proceedings of the 4th IEEE International Conference on Peer-to-Peer Computing*, Zurich, Switzerland, August 2004. IEEE. To appear.
10. A. Montresor, M. Jelasity, and O. Babaoglu. Robust aggregation protocols for large-scale overlay networks. In *Proceedings of The 2004 International Conference on Dependable Systems and Networks (DSN)*, pages 19–28, Florence, Italy, June 2004. IEEE Computer Society.
11. J. M. Ottino. Engineering complex systems. *Nature*, 427:399, January 2004.

Autonomic Runtime System for Large Scale Parallel and Distributed Applications

Huoping Chen, Byoung uk Kim, Jingmei Yang, and Salim Hariri
Dept. of Electrical and Computer Engineering
University of Arizona, Tucson AZ
Email: hariri@ece.arizona.edu

Manish Parashar
Dept. of Electrical & Computer Engineering
Rutgers, The State University of New Jersey
94 Brett Road, Piscataway, NJ 08854
Email: parashar@caip.rutgers.edu

Abstract

This paper presents a novel *autonomic* runtime system framework for addressing the fundamental challenges of scale and complexity inherent in emerging Grids and their applications. The autonomic framework is inspired by the approach used by biological systems to address similar challenges of scale and complexity. It will be used to develop and deploy self-managing autonomic runtime solutions and application/system components that are capable of continuously monitoring, adapting, and optimizing behaviors based on the runtime state and dynamically changing requirements. The runtime solutions will enable scalable and robust realizations of scientific and engineering simulations that will achieve unprecedented scales and resolutions and will provide important insights into complex phenomena, enabling revolutionary advances in science and engineering. Our approach to implement this framework is based on a collection of autonomous modules (e.g., agents, managers) that provide four basic services (self-management, self-optimizing, self-protecting and self-healing) that are essential to achieve the required autonomic operations. In this paper, we will describe the overall architecture of the autonomic runtime system and show how this system can self-optimize the computations of a large scale fire propagation simulations. Our initial results show significant performance gains can be achieved with the system exploit both the spatial and temporal characteristics of application states as well as the states of the underlying computing and communication resources.

1. Introduction

The Grids introduce a new set of challenges due to their scale and complexity. These systems will be constructed hierarchically with computational, communication, and storage heterogeneity across its levels. In capacity mode, these systems will be shared amongst users, resulting in highly dynamic runtime behaviors due to the sharing of processors, memories, and communication channels. Most currently used programming models and system software build on the abstraction of flat processor topologies and flat memories, resulting in intractable programming complexities, unmanageable codes, and applications that can achieve only a fraction of the system's peak performance. Furthermore, as the system approach petascales, the probability of failure of system components during the lifetime of an application quickly becomes a reality. As a result, reliability and fault tolerance become an increasing concern.

In addition to the system challenges**,** multiphysics simulation codes and the phenomena they model are similarly large complex, multi-phased, multi-scale, dynamic, and heterogeneous (in time, space, and state). They implement various numerical algorithms, physical constitutive models, domain discretizations, domain partitioners, communication/interaction models, and a variety of data structures. Codes are designed with parameterization in mind, so that numerical experiments may be conducted by changing a small set of inputs. The choices of algorithms and models have performance implications which are not typically known a priori. Advanced adaptive solution techniques, such as variable step time integrators and adaptive mesh refinement, add a new dimension to the complexity - the application realization changes as the simulation proceeds. This dynamism poses a new set of application development and runtime management challenges. For example, component behaviors and their compositions can no longer be statically defined. Further, their performance characteristics can no longer be derived from a small synthetic run as they depend on the state of the simulations and the underlying system. Algorithms that worked well at the beginning of the simulation become suboptimal as the solution deviates from the space the algorithm was optimized for.

The application and system challenges outlined above represent a level of complexity, heterogeneity, and dynamism that is rendering our programming environments and infrastructure brittle, unmanageable, and insecure. This, coupled with real-life constraints and the mathematical and scientific intricacies of future simulation codes (an increase in computing power will engender a commensurate increase in the expectations of what might be achieved with them), indicates that there needs to be a fundamental change in how these applications are formulated and managed on a large system. It is this fundamental change that we seek to identify and realize in this proposed research.

This paper presents a novel Autonomic Runtime System (ARS) to address the challenges of scale and complexity that is based on using autonomous agents for continuous runtime monitoring, estimation, analysis, and adaptation. The ARS provides the required key services to support autonomic control and management of Grid applications and systems. Specifically, ARS services include application/system monitoring and modeling, policy definition and enforcement, dynamic application composition, adaptation, fault management, and optimization.

The organization of the paper is as follows. In Section 2, we give a brief overview of related works. In Section 3, we discus the architecture of the ARS and the algorithms to implement its main services. In Section 4, we discuss the fire propagation simulation that will be used to benchmark and evaluate the performance gains that can be achieved by using the ARS self-optimization service. In Section 5, we discuss the experimental environment and results. In Section 6, we summarize the paper and discuss future research direction.

1. Previous Studies and Related Work

The related efforts [1,2,6,7,14,24,25,33,34,37,38,44] can be grouped into the following two areas of (a) autonomic frameworks and adaptive runtime systems, and (b) component-based software for large parallel scientific simulations are discussed below.

Autonomic Frameworks and Adaptive Runtime Systems: The prototype ***Autonomia*** [3,16] and ***AutoMate*** [4,5,21,23,28,29,30,39,40,45,46] autonomic runtimes are proof-of-concept prototypes for the autonomic runtime system that will be developed as part of this research. These systems demonstrate, both, the feasibility and the effectiveness of the autonomic approach. Autonomia provides an agent-based framework of autonomic management while AutoMate provides key services for the autonomic formulation, composition, and runtime management of large applications. The proposed research will build on these experiences and will develop and deploy an integrated autonomic runtime for petascale systems.

The ***Pragma*** [11,20,41,42,47,48] and ***ARMaDA*** [8,9,12,13] adaptive runtime management, partitioning, and load-balancing frameworks developed by the PIs support proactive and reactive management of dynamic (adaptive mesh refinement) applications. The systems provide capabilities for runtime system characterization and abstraction, application characterization, active control, and policy definition. Reactive system partitioning [41] uses system state to select and configure distribution strategies and parameters at runtime. Application aware partitioning [9,12,13,42,43] uses current runtime state to characterize the adaptive application in terms of its computation/communication requirements, its dynamics, and the nature of adaptations, and selects and configures the appropriate partitioner that matches current application requirements. Proactive runtime partitioning [47] strategies are based on performance prediction functions and estimate the expected performance of a particular application distribution.

These systems will be used to define policies for autonomic runtime management as well as to construct a suite of autonomic self-optimizing computational components. ***GridARM*** [10,18,19] is a prototype of such an autonomic partitioning framework. It optimizes the performance of structured adaptive mesh refinement (SAMR) applications. The framework has 3 components: (1) services for monitoring resource capabilities and application dynamics and characterizing the monitored state into natural regions – i.e., regions with relatively uniform structures and requirements; (2) deduction engine and objective function that define the appropriate optimization strategy based on runtime state and policies; and (3) the autonomic runtime manager that is responsible for hierarchically partitioning, scheduling, and mapping application working-sets onto virtual resources, and tuning application execution within the Grid environment. The adaptation schemes within GridARM include application aware partitioning [9], adaptive hierarchical partitioning [20], system sensitive partitioning [41], architecture sensitive communication mechanisms, and workload sensitive load balancing. GridARM has been shown to significantly improve applications' performance [10].

Autonomic Component/Object Framework: DIOS++ [22,32] is an infrastructure for enabling rule-based autonomic adaptation and control of distributed scientific applications and is a conceptual prototype of the Accord autonomic component architecture that will be developed as part of this proposal. DIOS++ provides (1) abstractions for enhancing existing application objects with sensors and actuators for runtime interrogation and control, (2) a control network that connects and manages the distributed sensors and actuators, and enables external discovery, interrogation, monitoring and manipulation of these objects at runtime, and (3) a distributed rule engine that enables the runtime definition, deployment, and execution of rules for autonomic application management. The DIOS++ framework, along with the Discover [15,17,26,27,31,32,35,36] computational collaboratory, is currently being used to enable autonomic monitoring and control of a wide range of scientific applications including oil reservoir, compressible turbulence, and numerical relativity simulations.

3. Autonomic Runtime System Architecture

The Autonomic Runtime System (ARS) exploits the temporal and heterogeneous characteristics of the scientific computing applications, and the architectural characteristics of the petascale computing and storage resources available at runtime to achieve high-performance, scalable, and robust scientific simulations that previously were not possible or ever attempted. ARS provides appropriate control and management services to deploy and configure the required software and hardware resources to run autonomically (e.g., self-optimize, self-heal) large scale scientific Grid applications. The ARS architecture is shown in Figure 1.

The ARS can be viewed as an application-based operating system that provides applications with all the services and tools required to achieve the desired autonomic behaviors (self-configuring, self-healing, self-optimizing, and self-protection). A proof of concept prototype of ARS has been successfully developed by the PIs involved in this project. The primary modules of ARS are the following:

Application Information and Knowledge (AIK) Repository*:* The AIK repository stores the application information status, the application requirements, and knowledge about optimal management strategies for both applications and system resources that have proven to be successful and effective. In addition, AIK contains the *Component Repository (CR)* that stores the components that are currently available for the users to compose their applications.

Event Server*:* The Event server receives events from the component managers that monitor components and systems and then notifies the corresponding engines subscribed to these events.

Autonomics Middleware Services *(Self-Configuration, Self-Optimization, Self-healing, etc.):* These runtime services maintain the autonomic properties of applications and system resources at runtime. To simplify the control and management tasks, we dedicate one runtime service for each desired attribute or functionality such as self-healing, self-optimizing, self-protection, etc. The event series notifies the appropriate runtime service whenever its events become true.

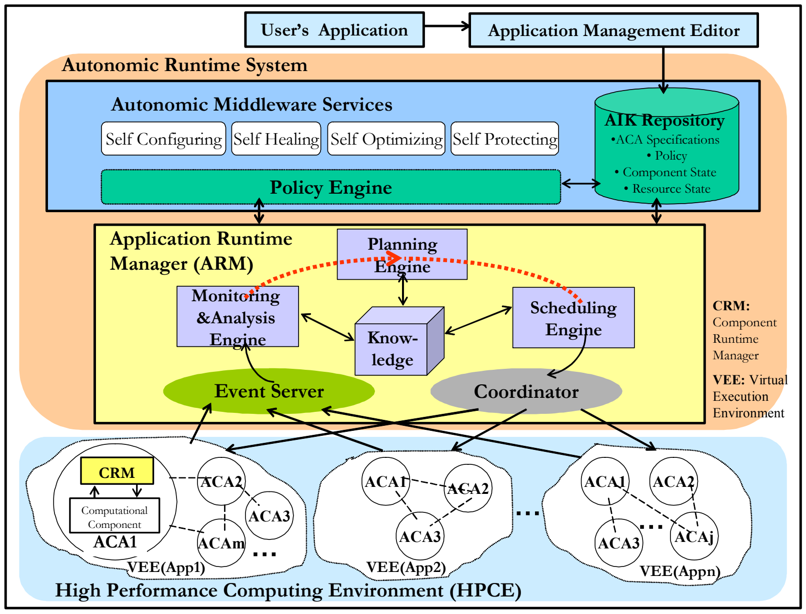


Figure 1. Autonomic Runtime System

Monitoring Service*:* There are two kinds of monitoring services: Resource Monitoring Service and Component Monitoring Service. *Resource Monitoring Service (RMS)* monitors the workload information and performance metrics at system level, which includes three parts: CPU/memory information, disk I/O information, and network information. *Component Monitoring Service (CMS)* defines a general interface for the autonomic components. Through the interface, the component can expose its status data such as execution time and current state as defined by the autonomic component architecture to CMS.

Application Runtime Manager (ARM): The ARM performs online monitoring to collect the status and state information using the component sensors. It analyzes component behaviors and when detecting any anomalies or state changes (for example, degradation in performance, component failure), ARM takes the appropriate control and management actions as specified in its control policy.

Autonomic Middleware Services (AMS)

The AMS provides four important services: Self-configuring, Self-optimizing; Self-protecting; and Self-healing. In this paper, we will focus on the self-optimizing service. The overall self-management algorithm is shown in Figure 2. The state of each active component is monitored by the Component Runtime Manager (CRM) monitors to determine if there is any severe deviation from the desired state (steps 1-3). When an unacceptable change occurs in the component behavior, CRM generates an event into the Event Server, which notifies ARM (step 4-6). Furthermore, CRM analyzes the event and determines the appropriate plan to handle that event (step 7 and step 8) and then executes the appropriate self-management routines (steps 9-10). However, if the problem cannot be handled by CRM, the ARM is invoked to take the appropriate management functions (steps 12-13) at a higher granularity (e.g., migrate the components to another machine due to failure or degradation in performance).

The self-management algorithm defines the main activities that must be performed to achieve autonomic behaviors of applications as well as system resources. In what follow, we describe the development the self-optimization service and use the fire simulation as a running example.

```
1 While (Component $ACA_i$ is running) do
2    State = $CRM_i$ Monitoring ($ACA_i$)
3    State_Deviation = State_Compare(State, DESIRED_STATE)
4    If (state_deviation == TRUE)
5            CRMi Send_Event(State)
6            Event Server Notify ARM
7            Event_Type = CRM_Analysis (State)
8            If ($CRM_{i\ ABLE)}$ Then
8                    Actions = CRM_Planning(State, Event_Type)
9                    Autonomic_Service $AS_j \in \{AS_{config}, AS_{heal}, AS_{optimization}, AS_{security}\}$
10                   Execute $AS_j$ (Actions)
11           Else
12                   Actions = ARM_Analysis (State, Event_Type)
13                   Execute $As_j$ (Actions)
14           EndIf
15   EndIf
16 EndWhile
```

Figure 2: Self Management Algorithm

3.1 Forest Fire Simulation – Running Example

The forest fire simulation model could be used to locate area of potential wildfires, predict the spread of wildfires based on both static conditions, such as vegetation conditions, and dynamic conditions, such as wind direction and speed, and help disaster coordinators to make the best use of available resources when fighting a wildfire. Our fire simulation model is based on fireLib [49] which is a C function library for predicting the spread rate and intensity of free-burning wildfires. It is derived directly from the BEHAVE [50] fire behavior algorithms for predicting fire spread in two dimensions, but is optimized for highly iterative applications such as cell- or wave-based fire growth simulation.

In our fire simulation model, the forest is represented as a 2-D cell-space composed of cells of dimensions l x b (l: length, b: breadth). For each cell there are eight major wind directions N, NE, NW, S, SE, SW, E, W as shown below. The weather and vegetation conditions are assumed to be uniform within a cell, but may vary in the entire cell space. A cell interacts with its neighbors along all the eight directions as shown below.

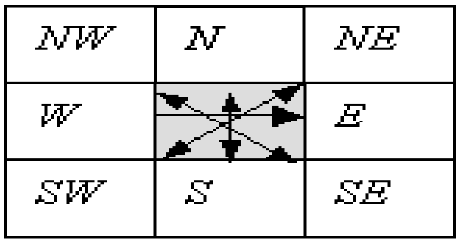


Figure 3: Fire directions after ignition

When a cell is ignited, its state will change from “unburned” to “burning”. During the “burning” phase, the fire will propagate to its eight neighbors along the eight directions. . The direction and the value of the maximum fire spread rate within the burning cell is computed using Rothermel's fire spread model [51], which takes into account the wind speed and direction, the vegetation type, the fuel moisture and terrain type, such as slope and aspect, in calculating the fire spread rate. The fire behavior in eight compass directions including the spread rate, the time for fire to spread to eight neighbors, and the flame length could be determined using the elliptical growth model developed by Anderson [52]. When the fire propagates to one neighbor, the neighbor cell will ignite and its state will change from "unburned" to "burning". With different terrain, vegetation and weather conditions, the fire propagation could form different spread patterns within the whole region.

We extended the sequential fire simulation program from [49] to work with multiple processors using MPI. This parallel version partitions the entire cell space among multiple processors and each processor works on its own part and exchange the necessary data with each other after each time step. Below is the pseudo code for the parallel version of our forest fire simulation:

```
1. All processors read their parts of cells from the partitioning data
2. Initialize the cell conditions including fuel bed and fuel particle characteristics, wind speed and
   direction, slope and aspect, and fuel moisture
3. Set up the first ignition cell
4. WHILE there are more cells that can ignite AND fire does not reach the edge of the whole space
        1) Set the current time with the next cell ignition time
        2) INCREMENT the time step
        3) FOR each cell within its part, each processor do
              a) IF the cell is not the next cell to ignite, THEN
                     Continue   //Skip this cell
                 END IF
              b) IF the cell is on the edge, THEN
                     Set the edge flag to 1 to indicate the fire has reached the edge
                 END IF
              c) Calculate the direction and the value of the maximum spread rate of the fire
                 within this cell;
              d) Change the cell state from "unburned" to "burning"
              e) FOR each neighbor of this cell
                     IF this neighbor's position exceed the whole space, THEN
                         Continue  //Skip this neighbor
                     END IF
                 END FOR
              f) Calculate the flame length and the time when fire spread to this neighbor
              g) IF the ignition time of this neighbor is greater than this fire spread time,
                 THEN
                        Update the ignition time of this neighbor with this fire spread time
                 END IF
              h) IF the next cell ignition time is greater than this fire spread time, THEN
                        Update the next cell ignition time with this fire spread time
                 END IF
           END FOR
        4) Each processor exchanges the ignition time and state changes of cells within its part
           with other processors and updates the ignition time of cells with the minimum value
        5) Each processor exchanges the next cell ignition time with other processors and
           updates it with the minimum value
        6) Each processor exchanges the edge flag with other processors and set it to 1
           whichever processor set it to 1
        7) IF it is output time, THEN
                  Output the cell state changes since last output time
           END IF
        8) IF the repartitioning flag is set, THEN
                   Read the repartitioning data
                  Unset the repartition flag
           END IF
   END WHILE
5. Save the ignition map and flame length maps to files
```

Figure 4. Parallel Forest Fire Simulation.

3.2 Self-Optimization Service

3.21 Online Monitoring and Analysis

There are two types of monitoring services. One collects information about the component/task state and another one to monitor the physical resource status such as CPU utilization, number of processes, available memory etc. Based on the current states of the application and the system resources, the analysis engine determine whether or not the current allocation of the applications components/tasks need to be changed. For example, if the current allocation leads to sever imbalance conditions among the processors running the application components, the online planning and scheduling is invoked to adopt the allocation in order to improve the application performance.

3.2.2 Online planning and Scheduling

The online planning engine partitions the forest fire simulation domain into two Natural Regions (NRs): Burning and Unburned regions where each region has the same temporal and spatial characteristics. In our planning algorithm, we use Z-curve to translate the two-dimension cell location into one dimension address to maintain the cell location adjacency. The Z-Curve is named for the 'Z' pattern that is traced when you access sequential objects from the database. An object Z-value is determined by interleaving the bits of its x and y coordinates, and then translating the resulting number into a decimal value. This new value is the index to be used to locate that object. Once the planning engine partitions the cells into two natural regions: Burning and Unburned regions and order them according to their z values such that the adjacent cells will be assigned to the same processor. The next step is to schedule these cells to processors by taking into consideration the state of the application (number of burning and unburned cells) and current load on the machines involved in the application execution.

The scheduling algorithm is shown in Figure 5. The scheduling algorithm will be triggered only when the load imbalance increases above a given threshold. In our approach, the online monitoring and analysis that monitors will set the partitioning flag once that event becomes true which in turn triggers the scheduling algorithm (Step 1 in the algorithm shown in Figure 5). In order to take into consideration both the system load (Step 2), and application state (number of burning and unburned cells (Step 3)). Once that is done, the execution time of a burning cell is estimated on each processor as well as the overall average execution time (Step 4). Based on these estimates, we can estimate the appropriate processor load ratio that will eliminate the imbalance conditions (Steps 5-6). Once the processor load ration (PLR) is determined, we use this ration to partition the current application load on all the processors (Step 7-8).

```
1 While Schedule_Flag is true Do
2     Get System State
              $N = Number\ of\ Processors$
              $System\_Load_i(t)$, $\forall i = 1,2,...,N$
3     Get Cell Application State ( $NR_B(t)$ , $NR_{UB}(t)$ )
          $NR_B(t) = \sum_{i=1}^{N} B\_Cell_i(t)$          // Number of burning cells running on processor i
              $NR_{UB}(t) = \sum_{i=1}^{N} UB\_Cell_i(t)$          // Number of unburned cells running on processor i
              $\forall i = 1,2,...,N$
4     Estimate Burning cell execution time on each processor ( $t_{B\_Cell_i}(t)$ ) and
      average execution time ( $t_{avg\_B\_Cell}(t)$ )
          $t_{B\_Cell_i}(t) = System\_Load_i(t) * t^1_{B\_Cell}$ ,
                  // $t^1_{B\_Cell}$ : Execution time of one burning cell with system load 1
          $t_{avg\_B\_Cell}(t) = \frac{\sum_{i=1}^{N} t_{B\_Cell\,i}(t)}{N}$,  $\forall i = 1,2,...,N$
5     Estimate Processor Allocation Factor (PAF)
          $PAF_i(t) = \frac{t_{avg\_B\_Cell}(t)}{t_{B\_Cell\,i}(t)}$ , $\forall i = 1,2,...,N$
6     Estimate Processor Load Ratio (PLR)
          $PLR_i(t) = \frac{PAF_i(t)}{\sum_{i=1}^{N} PAF_i(t)}$ , $\forall i = 1,2,...,N$
7     Estimate Application Partition Ratio (APR)
          $Application\_Load(t) = NR_B(t) + NR_{UB}(t)$
          $APR_i(t) = Application\_Load(t) * PLR_i(t)$ , $\forall i = 1,2,...,N$
8     For i = 1, to Number_of_processor do
              Assign the cells in $NR_B$ and $NR_{UB}$ according to $APR_i(t)$
      End For
  End While
```

Figure 5. Cell scheduling algorithm

4. Experimental Results

We are currently evaluating the performance of our algorithm on a cluster of workstations. The preliminary results indicate that a significant performance can be gained by exploiting the application and resources states at runtime.

References

[1] Agent Building and Learning Environment, http://www.alphaworks.ibm.com/tech/able, IBM.

[2] The Anthill project, http://www.cs.unibo.it/projects/anthill/index.html.

[3] AUTONOMIA: An Autonomic Computing Environment, http://www.ece.arizona.edu/~hpdc/-projects/AUTONOMIA/.

[4] Agarwal, M., V. Bhat, et al. (2003). AutoMate: Enabling Autonomic Applications on the Grid. Proceedings of the Autonomic Computing Workshop, 5th Annual International Active Middleware Services Workshop (AMS2003), Seattle, WA, IEEE Computer Society Press.

[5] Agarwal, M. and M. Parashar (2003). Enabling Autonomic Compositions in Grid Environments. Proceedings of the 4th International Workshop on Grid Computing (Grid 2003), Phoenix, AZ, IEEE Computer Society Press.

[6] Beynon, M. D., T. Kurc, et al. (2001). "Distributed Processing of Very Large Datasets with DataCutter." Journal of Parallel Computing 27(11): 1457-1478.

[7] Bramley, R., K. Chiu, et al. (2000). A Component Based Services Architecture for Building Distributed Applications. 9th IEEE International Symposium on High Performance Distributed Computing Conference, Pittsburgh, PA.

[8] Chandra, S. and M. Parashar (2001). An Evaluation of Partitioners for Parallel SAMR Applications. Proceedings of the 7th International Euro-Par Conference (Euro-Par 2001), Springer-Verlag.

[9] Chandra, S. and M. Parashar (2002). ARMaDA: An Adaptive Application-Sensitive Partitioning Framework for SAMR Applications. Proceedings of the 14th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2002), Cambridge, MA, ACTA Press.

[10] Chandra, S., M. Parashar, et al. (2003). GridARM: An Autonomic Runtime Management Framework for SAMR Applications in Grid Environments. Proceedings of the Autonomic Applications Workshop, 10th International Conference on High Performance Computing (HiPC 2003), Hyderabad, India, Elite Publishing.

[11] Chandra, S., S. Sinha, et al. (2002). Adaptive Runtime Management of SAMR Applications. Proceedings of the 9th International Conference on High Performance Computing (HiPC 2002), Bangalore, India, Springer-Verlag.

[12] Chandra, S., J. Steensland, et al. (2001). Adaptive Application Sensitive Partitioning Strategies for SAMR Applications. Denver, CO, Poster publication at IEEE/ACM Supercomputing 2001.

[13] Chandra, S., J. Steensland, et al. (2001). An Experimental Study of Adaptive Application Sensitive Partitioning Strategies for SAMR Applications. Proceedings of the 2nd LACSI (Los Alamos Computer Science Institute) Symposium 2001, Santa Fe, NM.

[14] Chaudhuri, S. AutoAdmin: Self-Tuning and Self-Administering Databases, http://research.microsoft.com/research/dmx/autoadmin, Microsoft Research Center.

[15] Chen, J., D. Silver, et al. (2003). Real Time Feature Extraction and Tracking in a Computational Steering Environment. Proceedings of the 11th High Performance Computing Symposium (HPC 2003), International Society for Modeling and Simulation, Orlando, FL.

[16] Hariri, S., L. Xue, et al. (2003). Autonomia: An Autonomic Computing Environment. Proceedings of the 2003 IEEE International Conference on Performance, Computing, and Communications.

[17] Jiang, L., H. Liu, et al. (2004). Rule-based Visualization in a Computational Steering Collaboratory. Proceedings of the International Workshop on Programming Paradigms for Grid and Metacomputing Systems, International Conference on Computational Science 2004 (ICCS 2004), Krakow, Poland.

[18] Khargharia, B., S. Hariri, et al. (2003). vGrid: A Framework for Development and Execution of Autonomic Grid Applications. Proceedings of the Autonomic Applications Workshop, 10th International Conference on High Performance Computing (HiPC 2003), Hyderabad, India, Elite Publishing.

[19] Khargharia, B., S. Hariri, et al. (2003). vGrid: A Framework for Building Autonomic Applications. Proceedings of the 1st International Workshop on Challenges for Large Applications in Distributed Environments (CLADE 2003), Seattle, WA, Computer Society Press.

[20] Li, X. and M. Parashar (2003). Dynamic Load Partitioning Strategies for Managing Data of Space and Time Heterogeneity in Parallel SAMR Applications. Proceedings of 9th International Euro-Par Conference (Euro-Par 2003), Klagenfurt, Austria, Springer-Verlag.

[21] Li, Z. and M. Parashar (2004). Rudder: A Rule-based Multi-agent Infrastructure for Supporting Autonomic Grid Applications. Proceedings of The International Conference on Autonomic Computing, New York, NY.

[22] Liu, H. and M. Parashar (2003). DIOS++: A Framework for Rule-Based Autonomic Management of Distributed Scientific Applications. Proceedings of the 9th International Euro-Par Conference (Euro-Par 2003), Lecture Notes in Computer Science, Klagenfurt, Austria, Springer-Verlag.

[23] Liu, H. and M. Parashar (2004). A Component Based Programming Framework for Autonomic Applications. Submitted to the International Conference on Autonomic Computing (ICAC-04).

[24] Lohman, G. M. and S. Lightstone (2002). SMART: Making DB2 (More) Autonomic. Very Large Data Bases Conference (VLDB'02).

[25] Lu, Q., M. Parashar, et al. (2003). "Parallel Implementation of Multiphysics Multiblock Formulations for Multiphase Flow in Subsurface." submitted for publication to Parallel Computing, Elsevier Publishers.

[26] Mann, V., V. Matossian, et al. (2001). "DISCOVER: An Environment for Web-based Interaction and Steering of High-Performance Scientific Applications." Concurrency and Computation: Practice and Experience 13(8-9): 737-754.

[27] Mann, V. and M. Parashar (2002). "Engineering an Interoperable Computational Collaboratory on the Grid." Concurrency and Computation: Practice and Experience, Special Issue on Grid Computing Environments 14(13-15): 1569-1593.

[28] Matossian, V., V. Bhat, et al. (2003). "Autonomic Oil Reservoir Optimization on the Grid." accepted for publication in Concurrency and Computation: Practice and Experience, John Wiley and Sons.

[29] Matossian, V. and M. Parashar (2003). Autonomic Optimization of an Oil Reservoir using Decentralized Services. Proceedings of the 1st International Workshop on Challenges for Large Applications in Distributed Environments (CLADE 2003), Seattle, WA, Computer Society Press.

[30] Matossian, V. and M. Parashar (2003). Enabling Peer-to-Peer Interactions for Scientific Applications on the Grid. Proceedings of the 9th International Euro-Par Conference (Euro-Par 2003), Klagenfurt, Austria, Springer-Verlag.

[31] Muralidhar, R., S. Kaur, et al. (2000). An Architecture for Web-based Interaction and Steering of Adaptive Parallel/Distributed Applications. Proceedings of the 6th International Euro-Par Conference (Euro-Par 2000), Springer-Verlag.

[32] Muralidhar, R. and M. Parashar (2003). "A Distributed Object Infrastructure for Interaction and Steering." Concurrency and Computation: Practice and Experience, Special Issue Euro-Par 2001 15(10): 957-977.

[33] Parashar, M. and J. C. Browne (2000). "System Engineering for High Performance Computing Software: The HDDA/DAGH Infrastructure for Implementation of Parallel Structured Adaptive Mesh Refinement." IMA: Structured Adaptive Mesh Refinement (SAMR) Grid Methods 117: 1-18.

[34] Parashar, M., J. C. Browne, et al. (1997). A Common Data Management Infrastructure for Parallel Adaptive Algorithms for PDE Solutions. Proceeding of Supercomputing '97 (ACM Sigarch and IEEE Computer Society), San Jose, CA, online publication, IEEE Computer Society Press.

[35] Parashar, M., G. v. Laszewski, et al. (2002). "A CORBA Commodity Grid Kit." Grid Computing Environments, Special Issue of Concurrency and Computations: Practice and Experience 14(13-15): 1057-1074.

[36] Parashar, M., R. Muralidhar, et al. "Enabling Interactive Oil Reservoir Simulations on the Grid." Concurrency and Computation: Practice and Experience, John Wiley and Sons.

[37] Parashar, M., J. A. Wheeler, et al. (1997). A New Generation EOS Compositional Reservoir Simulator: Framework and Multiprocessing. Proceedings of the Society of Petroleum Engineers, Reservoir Simulation Symposium, Paper No. 37977, Dallas, TX.

[38] Parashar, M. and I. Yotov (1998). An Environment for Parallel Multi-Block, Multi-Resolution Reservoir Simulations. Proceedings of the 11th International Conference on Parallel and Distributed Computing Systems (PDCS 98), Chicago, IL, International Society for Computers and their Applications (ISCA).

[39] Schmidt, C. and M. Parashar (2003). Flexible Information Discovery in Decentralized Distributed Systems. Proceedings of the 12th International Symposium on High Performance Distributed Computing, Seattle, WA, IEEE Computer Society Press.

[40] Schmidt, C. and M. Parashar (2003). A Peer to Peer Approach to Web Service Discovery. Accepted for publication Proceedings of the 2003 International Conference on Web Service (ICWS '03), Las Vegas, NV, Computer Science Research, Education and Applications (CSREA).

[41] Sinha, S. and M. Parashar (2002). "Adaptive System-Sensitive Partitioning of AMR Applications on Heterogeneous Clusters." Cluster Computing: The Journal of Networks, Software Tools, and Applications 5(4): 343-352.

[42] Steensland, J., S. Chandra, et al. (2002). "An Application-Centric Characterization of Domain-Based SFC Partitioners for Parallel SAMR." IEEE Transactions on Parallel and Distributed Systems 13(12): 1275-1289.

[43] Steensland, J., M. Thune, et al. (2000). Characterization of Domain-based Partitioners for Parallel SAMR Applications. Proceedings of the IASTED International Conference on Parallel and Distributed Computing Systems (PDCS'00), Las Vegas, NV, ACTA Press.

[44] Wang, P., I. Yotov, et al. (1997). A New Generation EOS Compositional Reservoir Simulator: Formulation and Discretization. Proceedings of the Society of Petroleum Engineers, Reservoir Simulation Symposium, Paper No. 37979, Dallas, TX.

[45] Zhang, G. and M. Parashar (2003). Dynamic Context-aware Access Control for Grid Applications. Proceedings of the 4th International Workshop on Grid Computing (Grid 2003), Phoenix, AZ, IEEE Computer Society Press.

[46] Zhang, G. and M. Parashar (2004). Context-aware Dynamic Access Control for Pervasive Applications. Accepted for publication in the Proceedings of the Communication Networks and Distributed Systems Modeling and Simulation Conference (CNDS 2004), 2004 Western MultiConference (WMC), San Diego, CA.

[47] Zhang, Y., J. Yang, et al. (2004). Autonomic Proactive Runtime Partitioning Strategies for SAMR Applications. Proceedings of the NSF Next Generation Software Systems (NGS) Program Workshop, held in conjunction with IPDPS 2004, Santa Fe, NM.

[48] Zhu, H., M. Parashar, et al. (2003). Self Adapting, Self Optimizing Runtime Management of Grid Applications Using PRAGMA. Proceedings of the NSF Next Generation Systems Program Workshop, IEEE/ACM 17th International Parallel and Distributed Processing Symposium, Nice, France, IEEE Computer Society Press.

[49] http://www.fire.org

[50] Andrews, Patricia L. BEHAVE: fire behavior prediction and fuel modeling system – BURN Subsystem, part 1. General Technical Report INT-194. Ogden, UT: U.S. Department of Agriculture, Forest Service, Intermountain Research Station; 1986. 130 p.

[51] Rothermel, Richard C. A mathematical model for predicting fire spread in wildland fuels. Research Paper INT-115. Ogden,UT: U.S. Department of Agriculture, Forest Service, Intermountain Forest and Range Experiment Station; 1972. 40 p.

[52] Anderson, Hal E. Predicting wind-driven wildland fire size and shape. Research Paper INT-305. Ogden,UT: U.S. Department of Agriculture, Forest Service, Intermountain Forest and Range Experiment Station; 1983. 26 p.

Generative Programming from a DSL Viewpoint

Charles Consel

INRIA/LaBRI
ENSEIRB – 1, avenue du docteur Albert Schweitzer,
Domaine universitaire - BP 99
33402 Talence Cedex, France
consel@labri.fr http://compose.labri.fr

1 Introduction

A domain-specific language (DSL) is typically created to model a program family [1]. The commonalities found in the target program family suggest abstractions and notations that are domain specific. In contrast with general-purpose languages (GPL), a DSL is readable for domain experts, often concise, and usually declarative. As an illustration, consider a program family aimed to communicate data in a distributed heterogeneous system. Such a layer is commonly needed in a variety of distributed applications and relies on a mechanism like the Sun Remote Procedure Call (RPC) [2]. The XDR layer of the Sun RPC consists of marshaling and un-marshaling both arguments and returned value to/from a machine-independent format. The program family, represented by all the possible (un-)marshaling variations, has lead to the development of a DSL that allows a programmer to concisely express the type of the remote procedure arguments and returned value, and to obtain the corresponding marshaling layer on both the client and server sides.

From a DSL viewpoint, generative programming [3] provides a variety of approaches and techniques to produce and optimize a DSL implementation such as the marshaling layer in the XDR case.

Outline. Section 2 discusses how generative tools can be used to compile DSL programs into GPL programs, from a DSL interpreter. When compiled into a GPL, a DSL program can be processed by existing generative tools for various purposes, including optimization, instrumentation and verification. In this context, the generative tools are driven by domain-specific information that is translated into different forms: declarations (Section 3), annotations (Section 4), and meta-programs (Section 5). In essence, these forms enable a DSL to be interfaced with existing generative tools.

2 High-Level Compilation

Some level of compilation can be achieved by specializing an interpreter with respect to a DSL program. Traditionally, this approach has been used to generate compilers from denotational-style language definitions [4, 5]. This approach was later promoted by Consel and Marlet in the context of DSLs, where the language user base often forbids major compiler development. Furthermore, the high-level nature of DSLs facilitates the introduction of optimizations.

An example of such a compilation strategy was used for a DSL, named Plan-P, aimed to specify application-specific protocols (*e.g.,* stream-specific degradation policies) to be deployed on programmable routers [6]. Program specialization was used at run time to achieve the effect of a *Just In Time* compiler, by specializing the Plan-P interpreter with respect to a Plan-P program. The resulting compiled programs run up to 50 times faster than their interpreted counterparts [6]. Importantly, such late compilation process enabled the safety and security of programs to be checked at the source level by the programmable routers, before being deployed. The use of specialization allowed to achieve late compilation without requiring any specific development, besides writing the interpreter.

At a lower level, meta-programming provides an alternative approach to deriving a compiler from an interpreter [7, 8]. This approach involves a careful annotation, and sometimes re-structuring, of the interpreter.

3 From a DSL Program to Declarations

A key feature of the DSL approach is to make domain-specific information an integral part of the programming paradigm. As such the programmer can be viewed as being prompted by the language to provide domain-specific information. This information may take the form of domain-specific types, syntactic constructs and notations. This language enrichment over GPLs is typically geared towards collecting sufficient information to make some domain-specific properties decidable [9] and thus to enable domain-specific verifications and optimizations. The collection of information may be achieved by dedicated program analyses, which are, by design of the DSL, simpler than the ones developed for GPLs.

Because the scope of computations to be expressed by a DSL is usually narrow, GPL constructs and operations are restricted or excluded. Furthermore this language narrowing may also be necessary to enable key properties to be statically determined. In fact, a DSL is commonly both a restricted and an enriched version of a GPL.

Once key properties are exhibited, the DSL program can be compiled into a GPL program. To retain domain-specific information, the generated program needs to be accompanied by some form of declarations specifying its properties. Of course, the declarations are tailored to a set of verification and/or optimization tools.

In the XDR case, an XDR description often defines fixed-size RPC arguments. When this description is compiled into GPL code, it can be accompanied by declarations aimed to drive some transformation tool. This situation is illustrated by the XDR compiler that conventionally generates C code gluing calls to a generic library. We have modified the XDR compiler to generate binding time of RPC argument sizes, besides marshaling code. As a result, a declaration is attached to the marshaling code of each data item to be transmitted. This declaration defines the size parameter as static, if it corresponds to a data item that has a fixed size; it is dynamic otherwise[1]. In this work, the generated declarations are targeted for a program specializer for C, named Tempo [10]. This tool performs a number of optimizations on both the marshaling code and the generic XDR library, including removal of buffer overflow checks and collapsing of function layers.

The XDR case is interesting because it demonstrates that, although a DSL introduces a new programming paradigm, it can still converge with and reuse GPL technology. Additionally, because the existing tool is often used in a narrower context, the results may be more predictable. In the XDR case for instance, since the compilation schemas, the XDR library, and the specialization contexts are fixed, specialization is fully predictable. This situation obviously does not exist for an arbitrary program with an arbitrary specialization context.

Aspect-oriented declarations could also be generated from a DSL program. For example, one could imagine adding a data compression phase to marshaling methods when invoked with data greater than a given size. In this case, the pointcut language has to be expressive enough to enable any program point of interest to be designated to insert the compression phase. Otherwise, annotations need to be injected in the generated program.

4 From a DSL Program to Annotations

Instead of generating a GPL program together with declarations, annotations can be directly inserted into the generated program. This strategy allows information to be accurately placed in the program. Like declarations, these annotations are geared towards specific tools. They can either be processed at compile time or run time.

At compile time, annotations can be used to guide the compilation process towards improving code quality or code safety. In the XDR case, for example, one could imagine a library where there would be two sets of buffer operations, with or without overflow checks. The selection of an operation would depend on annotations inserted in the program.

At run time, annotations can trigger specific actions upon run-time values. For instance, a data of an unknown size could be tested before being transmitted and be compressed if it is larger than a given threshold.

5 From a DSL Program to Meta-programming

A DSL program can also benefit from meta-programming technology. In this context, the compilation of a DSL program produces a code transformer and

[1] Note that these declarations go beyond data sizes. A detailed description of the language of specialization declarations and its application to the XDR example can be found elsewhere [10, 11].

code fragments. For example, in a multi-stage language like MetaOCaml [7], the code transformer consists of language extensions that enable the concise expression of program transformations.

In the XDR case, multi-stage programming would amount to generate code that expects some data size and produces code optimized for that size. Like program specialization, multi-stage programming corresponds to generic tools and can be directly used by a programmer. In the context of DSLs, existing tools can implement a specific phase of an application generator.

6 Conclusion

We examined generative programming approaches and techniques from a DSL viewpoint. We showed that a DSL can make use of these approaches and techniques in very effective ways. In essence, the DSL approach exposes information about programs that can be mapped into the realm of generative programming and be translated into declarations or annotations, which would normally be provided by a programmer. This situation illustrates the high-level nature of the DSL approach.

References

1. Consel, C.: From A Program Family To A Domain-Specific Language. Number 3016 in Lecture Notes in Computer Science, State-of-the-Art Survey. In: Domain-Specific Program Generation; International Seminar, Dagstuhl Castle. Springer-Verlag (2004) 19–29
2. Sun Microsystem: NFS: Network file system protocol specification. RFC 1094, Sun Microsystem (1989)
3. Czarnecki, K., Eisenecker, U.: Generative Programming. Addison-Wesley (2000)
4. Consel, C., Danvy, O.: Tutorial notes on partial evaluation. In: Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages, Charleston, SC, USA, ACM Press (1993) 493–501
5. Jones, N., Gomard, C., Sestoft, P.: Partial Evaluation and Automatic Program Generation. International Series in Computer Science. Prentice-Hall (1993)
6. Thibault, S., Consel, C., Muller, G.: Safe and efficient active network programming. In: 17th IEEE Symposium on Reliable Distributed Systems, West Lafayette, Indiana (1998) 135–143
7. Taha, W.: A Gentle Introduction to Multi-stage Programming. Number 3016 in Lecture Notes in Computer Science, State-of-the-Art Survey. In: Domain-Specific Program Generation; International Seminar, Dagstuhl Castle. Springer-Verlag (2004) 30 – 50
8. Czarnecki, K., O'Donnell, J.T., Striegnitz, J., Taha, W.: DSL Implementation in MetaOCaml, Template Haskell, and C++. Number 3016 in Lecture Notes in Computer Science, State-of-the-Art Survey. In: Domain-Specific Program Generation; International Seminar, Dagstuhl Castle. Springer-Verlag (2004) 51 – 72
9. Consel, C., Marlet, R.: Architecturing software using a methodology for language development. In Palamidessi, C., Glaser, H., Meinke, K., eds.: Proceedings of the 10^{th} International Symposium on Programming Language Implementation and

Logic Programming. Number 1490 in Lecture Notes in Computer Science, Pisa, Italy (1998) 170–194

10. Consel, C., Lawall, J., Le Meur, A.F.: A tour of Tempo: A program specializer for the C language. Science of Computer Programming (2004)
11. Le Meur, A.F., Lawall, J., Consel, C.: Specialization scenarios: A pragmatic approach to declaring program specialization. Higher-Order and Symbolic Computation **17** (2004) 47–92

Generative Programming from an AOP/CBSE Perspective

Mira Mezini and Klaus Ostermann

Darmstadt University of Technology, Germany
{mezini,ostermann}@informatik.tu-darmstadt.de

1 Introduction

Every software system can be conceived from multiple different perspectives, resulting in different decompositions of the software into different ”domain-specific” types and notations. However, traditional programming languages do not explicitly support the construction of software as a ”superimposition” of co-existing independent partial models. They rather assume that real-world systems have “intuitive”, mind-independent, preexisting concept hierarchies, thus, putting emphasis on hierarchical modeling, which basically forces all aspects of a software system to be expressed in terms of the same set of concepts, or refinements thereof.

In this paper, we briefly illustrate the problems that follow from this assumption and discuss issues involved in solving them. Our position is that general-purpose languages (GPLs) – rather than domain-specific languages – with built-in support for expressing the interaction (superimposition) of independent partial models in an abstract way in accordance with the familiar principles of abstraction and information hiding are needed. We consider the variant of aspect-oriented programming in AspectJ as an excellent starting point in this direction but observe that more powerful abstraction mechanisms are needed. We distinguish between mechanisms for structural (concept) mappings between partial models and mechanisms for behavioral (control/data flow) mapping. Roughly speaking, AspectJ provides intertype declarations and pointcut/advice meachnisms for structural, respectively behavioral mapping. We discuss how the aspect-oriented language Caesar [3], advances AspectJ with respect to structural mapping and outline the problems with current mechanisms for behavioral mapping as well as ideas about how to solve these problems.

The reminder of this position paper is organized as follows. In Sec. 2, we define some terminology. Especially, we intuitively define the notion of crosscutting models versus hierarchical models. In Sec. 3, we outline issues to be solved for expressing the superposition between crosscutting models in an abstract way, discuss the mechanisms available for this purpose in Caesar, and outline some areas of ongoing work on increasing the abstraction of the behavioral mappings. In Sec. 4, we compare the usage of GPLs for crosscutting models with domain-specific languages and program generation techniques, as fostered by popular approaches like model-driven architecture (MDA). Sec. 5 concludes.

2 Crosscutting versus Hierarchical Models

The criteria which we choose to decompose software systems into modules has significant impact on the software engineering properties of the software. In [6] Parnas observed that a data-centric decomposition eases changes in the representation of data structures and algorithms operating on them. Following on Parnas work, Garlan et al. [2] argue that function-centric decomposition on the other side better supports adding new features to the system, a change which they show to be difficult with the data-centric decomposition.

Software decomposition techniques so far, including object-oriented decomposition, are weak at supporting multi-view decomposition, i.e., the ability to simultaneously breakdown the system into inter-related units, whereby each breakdown is guided by independent criteria. What current decomposition technology does well is to allow us to view the system at different abstraction levels, resulting in several *hierarchical* odels of it, with each model be a refined version of its predecessor in the abstraction levels.

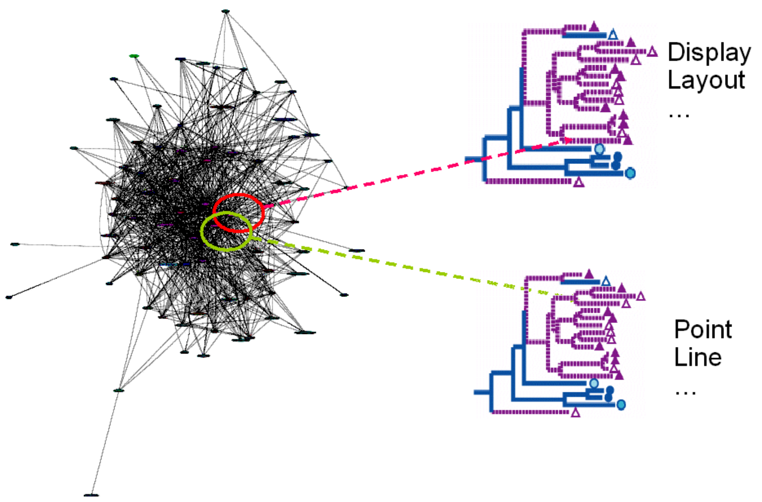


Fig. 1. Crosscutting models

By multi-view decomposition, we mean support for simultaneous *crosscutting* rather than *hierarchical* models. The key point is that our perception of the world depends heavily on the perspective from which we look at it: Every software system can be conceived from multiple different perspectives, resulting in different decompositions of the software into different "domain-specific" types and notations. In general, these view-specific decompositions are equally reasonable, none of them being a sub-ordinate of the others, and the overall definition of the system results from a superimposition of them.

The problem is that models resulting from simultaneous decomposition of the system according to different criteria are in general "crosscutting" with respect to the execution of the system resulting from their composition. With the conceptual framework used so far, *crosscutting* can be defined as a relation between two models with respect to the execution of the software described by the models. This relation if defined via *projections* of models (hierarchies).

A projection of a model M is a partition of the concern space into subsets $o_1, \ldots, o_n$ such that each subset o_i corresponds to a leaf in the model. Now, two models, M and M', are said to be crosscutting, if there exist at least two sets o and o' from their respective projections, such that, $o \cap\ o'$, and neither $o \subseteq o'$, nor $o' \subseteq o1$.

On the contrary, a model M is a *hierarchical refinement* of a model M' if their projections $o_1, \ldots, o_n$ and $o'_1, \ldots, o'_m$ are in a subset relation to each other as follows: there is a mapping $p : \{1, \ldots, n\} \rightarrow \{1, \ldots, m\}$ such that $\forall i \in \{1, \ldots, n\} : o_i \subseteq o'_{p(i)}$. Crosscutting models are themselves not the problem, since they are inherent in the domains we model. The problem is that our languages and decomposition techniques do not (properly) support crosscutting modularity (see the discussion on decomposition arbitrariness above).

In [4], we argue that even approaches with powerful hierarchical modularity, such as feature-oriented approaches [7, 1, 5] exhibit severe problems which we trace down to the lack of support for crosscutting modularity. Such approaches are superior to framework technology, due to their notion of a first-class layer module, which allows to define the delta pertaining to one feature into a single module that can be refined and used polymorphically just as classes can. However, they are incapable to express the interaction between a feature and the rest of the system in a modularized way. First, mapping the feature concepts onto existing concepts is a challenge when there is no one to one correspondence between the two. Second, it is not possible to express the interaction between a feature and the dynamic control flow of the rest of the system in a modular way abstracting away details of the control flow that are irrelevant for the interaction. The reason for this is that the "join points" that can be expressed are limited to individual method calls in the form of method overriding; there are no means to specify general sets of related join points that may crosscut the given module structure. We show that these problems seriously damage the scalability of the divide-and-conquer technique underlying FOAs.

One of the key observations of the aspect-oriented software development is that a programming technique that does not support simultaneous decomposition of systems along different criteria suffers from what we call *arbitrariness of the decomposition hierarchy* problem, which manifests itself as tangling and scattering of code in the resulting software, with known impacts on maintainability and extendibility. With a 'single-minded' decomposition technique that supports only hierarchical models, we have to choose one fixed classification sequence. However, the problem is that with a fixed classification sequence, only one concern is expressed concisely in terms of its inherent concepts whereas all other concerns are tangled in the resulting hierarchical structure.

3 Crosscutting Abstraction Mechanisms

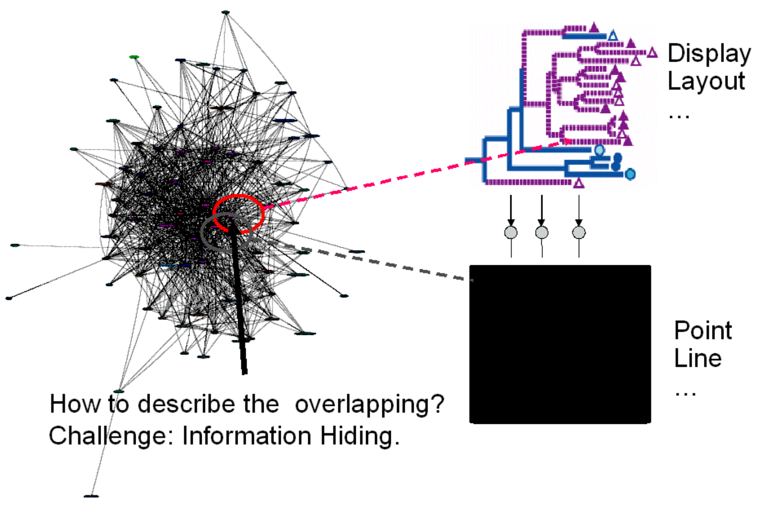

Display
Layout
...
Point
Line
...
How to describe the overlapping?
Challenge: Information Hiding.

Fig. 2. Information hiding and crosscutting models

Now, once we adopt a paradigm to software construction as a superimposition of different crosscutting models, the question is how to express this superimposition in a modular way and what abstractions are needed for the interface between crosscutting models. Fig. 2 is an attempt to illustrate the issue schematically.

The figure illustrates that we have two overlapping models of the same system. The tricky part is to describe how these two models interact with each other without referring too much to the implementation details of the models. This is illustrated by the black box with lollipops on top of it: We need a kind of *interface* to a crosscutting model that hides its implementation details.

There are two facets of expressing the overlap: (a) static (structural), and (b) dynamic, (behavioral) mapping. This is illustrated in Fig. 3, by means of two OO crosscutting models. In order to express how these two independent models interact in creating a whole, we need both to express how their concepts map to each other, illustrated by the arrows in the upper part of the figure, as well as how there control flows interact, illustrated by the lower part of Fig. **??**.

In [3, 4] we outline the deficiencies of AspectJ with respect to the first facet of expressing model superimposition. In [4], we argue that AspectJ is lacking a layer module concept as powerful as the one supported in feature-oriented approaches and discuss how the aspect-oriented language Caesar [3], we have been working on solves these problems.

In Caesar, a model is described by a bidirectional interface (see the `Pricing` interface in Fig. 4 for an illustration) that defines the abstractions in a particu-

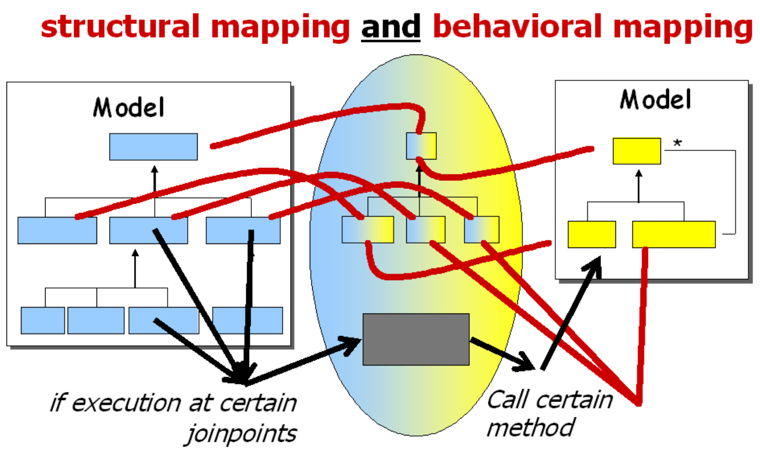

structural mapping and behavioral mapping
Model
Model
*
if execution at certain joinpoints
Call certain method

Fig. 3. Superimposing crosscutting models

lar domain. Different components can be implemented in terms of this domain model. Later on, such a model can be superimposed on an existing system by means of a so-called *binding*, which defines both a structural and a behavioral mapping in order to coordinate both worlds.

In [4], we argue that AspectJ is superior to FOAs for its sophisticated and powerful pointcut model that allows to express the behavioral mapping in a more precise and abstract way as it is possible with FOA. In contrast to the FOA solution, no shadowing is necessary in order to trigger the functionality of a feature in the base application. Pointcuts enable us to abstract over control flows. With more advanced mechanisms such as wildcards, field get/sets, `cflow`, etc., a pointcut definition also becomes more stable with respect to changes in the base structure than the corresponding set of overridden methods in FOA. The use of pointcuts instead of shadowing parts of an inherited base structure avoids the scalability problem mentioned in the FOA discussion. The key point is that with pointcuts we can abstract over details in the control flow that are irrelevant to the feature integration. Equivalent abstraction mechanisms are missing in FOAs.

In its current instantiation, Caesar has adopted the pointcut language of AspectJ. However, this language has problems. AspectJ-like languages come with a set of predefined pointcut designators, e.g., `call` or `get`, and pointcuts are not first-class values. To convey an intuition of what we mean by first-class pointcuts, let us consider identifying all setter join points were the value of a variable is changed that is read in the control flow of a certain method, `m`, the goal being that we would like to recall `m`, at any such point. Assuming a hypothetical AspectJ compiler that employs some static analysis techniques to predict control flows, one can write a pointcut `p1` that selects all getters in the predicted control flow of `m`. However, it is not possible to combine `p1` with another pointcut `p2` which takes the result of `p1` as a parameter, retrieves the names of the variables read

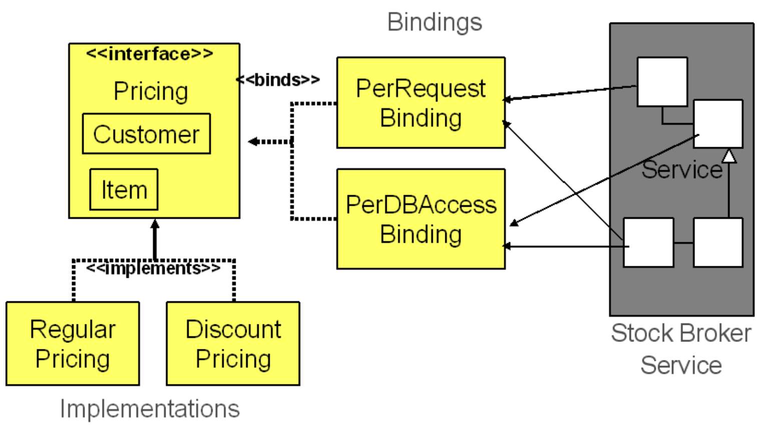


Fig. 4. Overview of Caesar concepts

in the join points selected by p1, and than selects the set of setter join points where one of these variables is changed. It is in fact, impossible to express the above semantics declaratively with language technology, in which pointcuts are processed in a single pass. What we need is the ability to reason about p1 and p2. This requires a reification of the result crosscut specified by p1, so that the names of the variables read by join points contained in it can be retrieved and than used in p2.

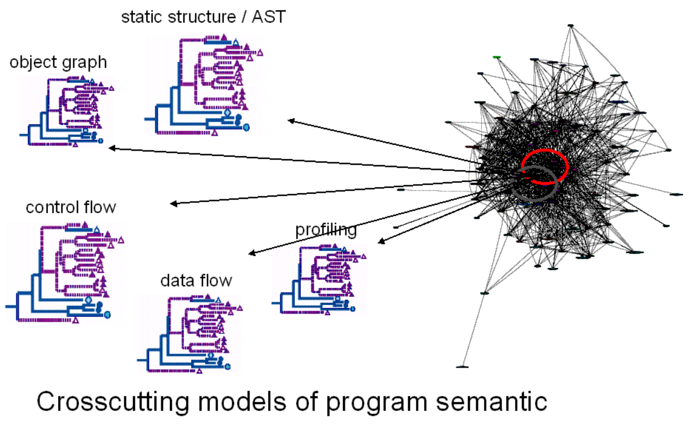


Fig. 5. Crosscutting models of program semantics

Our vision is that pointcuts should have a first-class status in an AOP platform; it should be possible to reason about a pointcut, and especially to define new pointcuts by reasoning about other pointcuts. We envision an AOP model in which pointcuts are sets of nodes in some representation of the program's semantics. Such sets are selected by queries on node attributes written in a query language and can be passed around to other query functions as parameters. These semantic models can be as diverse as abstract syntax treees, control flow graphs, data flow graphs, object graphs or profiling models (see Fig. 5).

4 General-Purpose or Program Generation?

In the context of model-driven architecture and other technologies, domain-specific languages (DSLs) and program generation techniques have gained considerable popularity. This techniques serve a similar goal as aspect-oriented programming, namely the combination of crosscutting models. Using DSLs and program generation, a model is encoded as a DSL (thereby fulfilling a similar purpose as bidirectional interfaces in Caesar). A program in the DSL (corresponding to components implementing a bidirectional interface) is combined with other models in a program generator, which produces the combined program in the form of generated sourcecode in a general-purpose language.

In a way, using this approach is easy and straightforward, because any semantics whatsoever can easily be encoded by manipulating and computing code. However, this approach also has some severe disadvantages:

- Supporting a new domain-specific model means writing a new program generator. Program generation is hard to understand, however. Instead of encoding the intention of the programmer directly, one has to think about the semantics of a program in terms of the program it generates. This additional "indirection" is a tremendous burden on the programmer. Besides, there are also many practical issues that make program generation painful, e.g., debugging, maintenance of generated code.
- Many features that have been available in conventional languages, e.g., control- and data structures or type-checking, have to be re-invented and re-implemented or are simply missing in DSLs.
- Basically all features that are added by the program generator (i.e., all crosscutting models) have to be known in advance, before writing the program generator. Writing a highly-configurable program generator makes the required effort even bigger. It is also a known fact that it is very hard to combine different program generators. Hence, extensionability with respect to adding additional crosscutting models in this approach is very limited.

Every general-purpose language allows the definition of domain-specific vocabulary by giving meaning to new names, e.g., functions, data structures etc. However, these "conventional" mechanisms to domain-specific models is obviously not sufficient, because otherwise program generation techniques would have never been so popular despite the disadvantages mentioned above. It is our position, however, that we should instead strive for new general-purpose abstraction mechanisms for crosscutting models that render the need for DSLs and program generators superfluous.

5 Summary

Traditional programming languages assume that real-world systems have "intuitive", mind-independent, preexisting concept hierarchies. This is in contrast to our perception of the world, which depends heavily on the context from which it is viewed. Since programmers want to describe "what" and not "how", we argued that programming languages should be augmented with abstraction mechanisms to encode and combine different crosscutting models instead of using program generation techniques to encode domain-specific knowledge.

References

1. D. Batory, V. Singhal, J. Thomas, S. Dasari, B. Geraci, and M. Sirkin. The genvoca model of software-system generators. *IEEE Software*, 11(5), 1994.
2. D. Garlan, G. E. Kaiser, and D. Notkin. Using tool abstraction to compose systems. *Computer*, 25(6):30–38, 1992.
3. M. Mezini and K. Ostermann. Conquering aspects with Caesar. In *Proc. International Conference on Aspect-Oriented Software Development (AOSD '03), Boston, USA*, 2003.
4. M. Mezini and K. Ostermann. Variability management with feature-oriented programming and aspects. In *Proceedings of FSE '04 (to appear)*, 2004.
5. K. Ostermann. Dynamically composable collaborations with delegation layers. In *Proceedings of ECOOP '02, LNCS 2374, Springer*, 2002.
6. D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.
7. Y. Smaragdakis and D. Batory. Implementing layered designs with mixin-layers. In *Proceedings of ECOOP '98*, pages 550–570, 1998.

Overview of Generative Software Development

Krzysztof Czarnecki

University of Waterloo, Canada
czarnecki@acm.org

1 Introduction

Object-orientation is recognized as an important advance in software technology, particularly in modeling complex phenomena more easily than its predecessors. But the progress in reusability, maintainability, reliability, and even expressiveness has fallen short of expectations. As units of reuse, objects have proven too small. Frameworks are hard to compose, and their development remains an art. Components offer reuse, but the more functional the component, the larger and less reusable it becomes. And patterns, while intrinsically reusable, are not an implementation medium.

Current research and practical experience suggest that achieving significant progress with respect to software reuse requires a paradigm shift towards modeling and developing software system families rather than individual systems. *System-family engineering* (also known as *product-line engineering*) seeks to exploit the commonalities among systems from a given problem domain while managing the variabilities among them in a systematic way [1–3]. In system-family engineering, new system variants can be rapidly created based on a set of reusable assets (such as a common architecture, components, models, etc.).[1] Frameworks and components are still useful as implementation technologies, but the scope and shape of reusable abstractions is determined and managed through a system-family approach.

Generative software development is a system-family approach, which focuses on automating the creation of system-family members: a given system can be automatically generated from a specification written in one or more textual or graphical *domain-specific languages* (DSLs) [1, 4–9]. A DSL can offer several important advantages over a general-purpose language:

– *Domain-specific abstractions*: a DSL provides pre-defined abstractions to directly represent concepts from the application domain.
– *Domain-specific concrete syntax*: a DSL offers a natural notation for a given domain and avoids syntactic clutter that often results when using a general-purpose language.

[1] System-family engineering is mainly concerned with building systems from common assets, whereas product-line engineering additionally considers scoping and managing common product characteristics from the market perspective. In order to be more general, this paper adheres to system-family terminology.

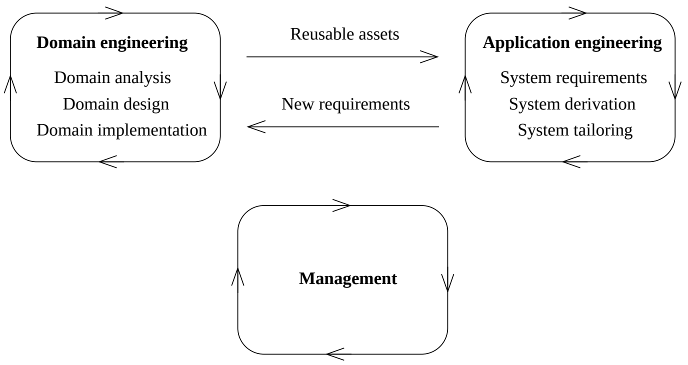


Fig. 1. Main processes in system-family engineering

– *Domain-specific error checking*: a DSL enables building static analyzers that can find more errors than similar analyzers for a general-purpose language and that can report the errors in a language familiar to the domain expert.
– *Domain-specific optimizations*: a DSL creates opportunities for generating optimized code based on domain-specific knowledge, which is usually not available to a compiler for a general-purpose language.
– *Domain-specific tool support*: a DSL creates opportunities to improve any tooling aspect of a development environment, including, editors, debuggers, version control, etc.; the domain-specific knowledge that is explicitly captured by a DSL can be used to provide more intelligent tool support for developers.

This paper gives an overview of the basic concepts and ideas of generative software development including domain and application engineering, generative domain models, networks of domains, and technology projections. The paper closes by discussing the relationship of generative software development to other emerging areas such as Model Driven Development and Aspect-Oriented Software Development.

2 Domain Engineering and Application Engineering

System family engineering distinguishes between at least two kinds of development processes: *domain engineering* and *application engineering* (see Figure 1). Typically, there is also a third process, *management*, but this paper focuses on the two development processes (for more information on process issues see [1,2]). Generative software development, as a system-family approach, subscribes to the process model in Figure 1, too.

Domain engineering (also known as *core asset development*) is "development for reuse". It is concerned with the development of reusable assets such as components, generators, DSLs, documentation, etc. Similar to single-system engineering, domain engineering also includes analysis, design, and implementation activities. However, these are focused on a class of systems rather than just a single system.[2] *Domain analysis* involves determining the scope of the family to be built, identifying the common and variable features among the family members, and creating structural and behavioral specifications of the family. *Domain design* covers the development of a common architecture for all the members of the system family and a plan of how individual systems will be created based on the reusable assets. Finally, *domain implementation* involves implementing reusable assets such as components, generators, and DSLs.

Application engineering (also referred to as *product development*) is "development with reuse", where concrete applications are built using the reusable assets. Just as traditional system engineering, it starts with requirements elicitation, analysis, and specification; however, the requirements are specified as a delta from or configuration of some generic system requirements produced in domain engineering. The requirements specification is the main input for *system derivation*, which is the manual or autmated construction of the system from the reusable assets.

Both processes feed on each other: domain-engineering supplies application engineering with the reusable assets, whereas application engineering feeds back new requirements to domain engineering. This is so because application engineers identify the requirements for each given system to be built and may be faced with requirements that are not covered by the existing reusable assets. Therefore, some amount of application-specific development or *tailoring* is often required in order to quickly respond to the customer's needs. However, the new requirements should be fed back into domain engineering in order to keep the reusable assets in sync with the product needs. Different models for setting up these processes in an organization, e.g., separate or joint product-development and domain-engineering teams, are discussed in [10].

Domain engineering can be applied at different levels of maturity. At minimum, domain analysis activities can be used to establish a common terminology among different product-development teams. The next level is to introduce a common architecture for a set of systems. Further advancement is to provide a set of components covering parts or all of the systems in the system family. Finally, the assembly of these components can be partially or fully automated using generators and/or configurators. The last level represents the focus of generative software development. In general, the generated products may also contain non-software artifacts, such as test plans, manuals, tutorials, maintenance guidelines, etc.

[2] Both terms "system family" and "domain" imply a class of systems; however, whereas the former denotes the actual set of systems, the latter refers more to the related area of knowledge. The use of the one or the other in compounds such as "domain engineering" is mostly historical.

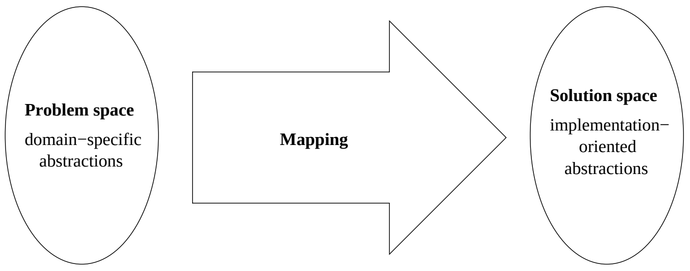

Problem space
domain−specific abstractions
Mapping
Solution space
implementation−oriented abstractions

Fig. 2. Mapping between problem space and solution space

3 Mapping Between Problem Space and Solution Space

A key concept in generative software development is that of a mapping between *problem space* and *solution space* (see Figure 2), which is also referred to as a *generative domain model*. Problem space is a set of domain-specific abstractions that can be used to specify the desired system-family member. By "domain-specific" we mean that these abstractions are specialized to allow application engineers to express their needs in a way that is natural for their domain. For example, we might want to be able to specify payment methods for an electronic commerce system or matrix shapes in matrix calculations. The solution space, on the other hand, consists of implementation-oriented abstractions, which can be instantiated to create implementations of the specifications expressed using the domain-specific abstractions from the problem space. For example, payment methods can be implemented as calls to appropriate web services, and different matrix shapes may be realized using different data structures. The mapping between the spaces takes a specification and returns the corresponding implementation.

There are at least two different views at the mapping between problem space and solution space in generative software development: *configuration view* and *transformational* view.

In the configuration view, the problem space consists of domain-specific concepts and their features (see Figure 3). The specification of a given system requires the selection of features that the desired system should have. The problem space also defines illegal feature combinations, default settings, and default dependencies (some defaults may be computed based on some other features). The solution space consists of a set of implementation components, which can be composed to create system implementations. A system-family architecture sets out the rules how the components can be composed. In the configuration view, an application programmer creates a configuration of features by selecting the desired ones, which then is mapped to a configuration of components. The mapping between both spaces is defined by construction rules (certain configurations of features translate into certain configurations of implementation components) and optimizations (some component configurations may have better non-functional properties then others). The mapping plus the illegal feature

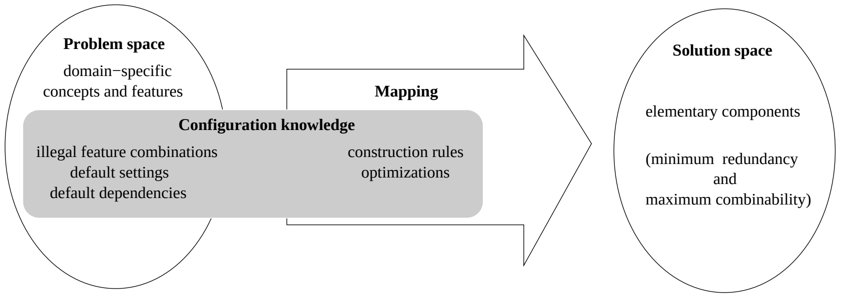


Fig. 3. Configuration view on the mapping between problem space and solution space

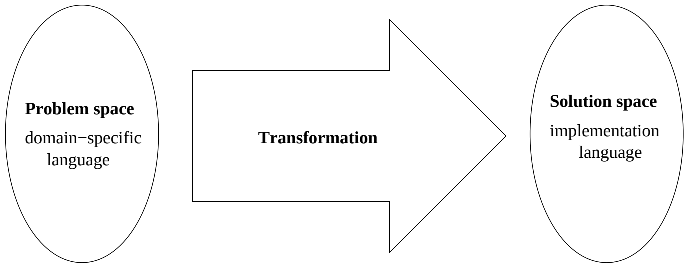


Fig. 4. Transformational view on the mapping between problem space and solution space

combinations, default settings, and default dependencies collectively constitute *configuration knowledge*. Observe that the separation between problem and solution space affords us the freedom to structure abstractions in both spaces differently. In particular, we can focus on optimally supporting application programmers in the problem space, while achieving reuse and flexibility in the solution space.

In the transformational view, a problem space is represented by a domain-specific language, whereas the solution space is represented by an implementation language (see Figure 4). The mapping between the spaces is a transformation that takes a program in a domain-specific language and yields its implementation in the implementation language. A domain-specific language is a language specialized for a given class of problems. Of course, the implementation language may be a domain-specific language exposed by another domain. The transformational view directly corresponds to the Draco model of domains and software generation [4].

Despite the superficial differences, there is a close correspondence between both views. The problem space with its common and variable features and constraints in the configuration view defines a domain-specific language, and the components in the solution space can also be viewed as an implementation language. For example, in the case of generic components, we can specify this target language as a GenVoca grammar with additional well-formedness constraints [6,11]. Thus, the configuration view can also be interpreted as a mapping between languages.

The two views relate and integrate several powerful concepts from software engineering, such as domain-specific languages, system families, feature modeling, generators, components, and software architecture. Furthermore, the translation view provides a theoretical foundation for generative software development by connecting it to a large body of existing knowledge on language theory and language translation.

4 Network of Domains

Observe that Figure 2 can be viewed recursively, i.e., someone's problem space may be someone else's solution space. Thus, we can have chaining of mappings (see Figure 5 a). Furthermore, a mapping could take two or more specifications and map them to one (or more) solution space (see Figure 5 b). This is common when different aspects of a system are represented using different DSLs. A mapping can also implement a problem space in terms of two or more solution spaces (see Figure 5 c). Finally, different alternative DSLs (e.g., one for beginners and one for expert users) can be mapped to the same solution space (see Figure 5 d), and the same DSL can have alternative implementations by mappings to different solution spaces (e.g., alternative implementation platforms; see Figure 5 e).

In general, spaces and mappings may form a hypergraph, which can even contain cycles. This graph corresponds to the idea of a *network of domains* by Jim Neighbors [4], where each implementation of a domain exposes a DSL, which may be implemented by transformations to DSLs exposed by other domain implementations.

5 Feature Modeling and Feature-Oriented Approach to Generative Software Development

Feature modeling is a method and notation to elicit and represent common and variable features of the systems in a system family. Feature modeling was first proposed by Kang et al in [12] and since then has been extended with several concepts, e.g., feature and group cardinalities, attributes, and diagram references [13].

An example of a feature model is shown in Figure 6. The model expresses that an electronic commerce system supports one or more different payment methods; it provides tax calculation taking into account either the street-level address, or postal code, or just the country; and it may or may not support shipment of physical goods. A feature diagram such as in Figure 6 may be supplemented with additional information including constraints (selecting a certain feature may require or exclude the selection of another feature), binding times (features may be intended to be selected at certain points in time), default attribute values

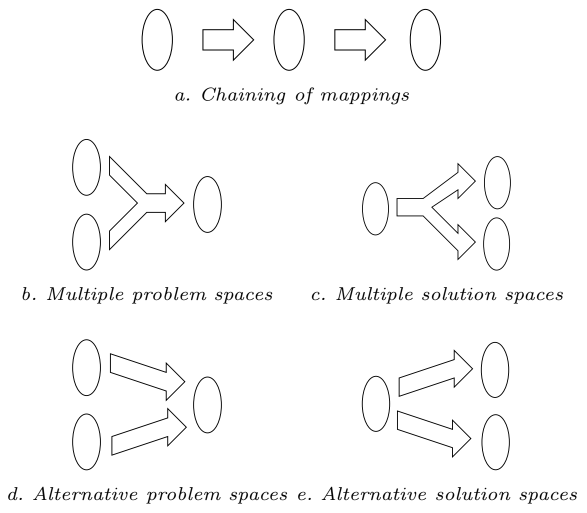

a. Chaining of mappings
b. Multiple problem spaces
c. Multiple solution spaces
d. Alternative problem spaces
e. Alternative solution spaces

Fig. 5. Different arrangements of mappings between problem and solution spaces

and default features, stakeholders interested in a given feature, priorities, and more. Features may or may not correspond to concrete software modules. In general, we distinguish the following four cases:

– *Concrete* features such as data storage or sorting may be realized as individual components.
– *Aspectual* features such as synchronization or logging may affect a number of components and can be modularized using aspect technologies.
– *Abstract* features such as performance requirements usually map to some configuration of components and/or aspects.
– *Grouping* features may represent a variation point and map to a common interface of plug-compatible components, or they may have a purely organizational purpose with no requirements implied.

Feature modeling gives rise to a feature-oriented approach to generative software developement. [6] In the early stages of software family development, feature models provide the basis for scoping a system family by recording and assessing information such as which features are important to enter a new market or remain in an existing market, which features incur a technological risk, what is the projected development cost of each feature, and so forth [14]. Subsequently, feature models created in domain analysis are the starting point in the development of both system-family architecture and DSLs (see Figure 7). Architecture development takes a solution-space perspective at the feature models: it concentrates on the concrete and aspectual features that need to be implemented as components and aspects. Familiar architectural patterns [15, 16] can be applied, but with the special consideration that the variation points expressed in

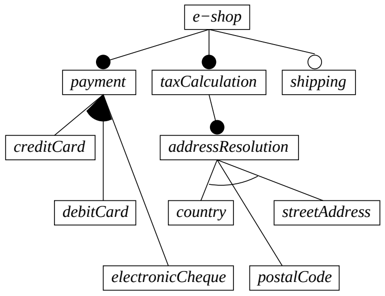


Fig. 6. Example of a feature diagram

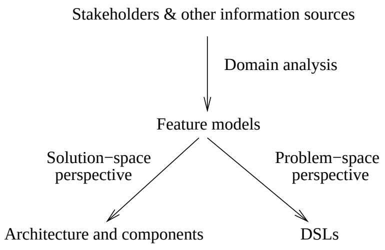

Stakeholders & other information sources
Domain analysis
Feature models
Solution−space perspective
Problem−space perspective
Architecture and components
DSLs

Fig. 7. Feature-oriented approach

the feature models need to be realized in the architecture. During subsequent DSL development, a problem-space perspective concentrating on features that should be exposed to application developers determines the required DSL scope, possibly requiring additional abstract features.

6 Technology Projections and Structure of DSLs

Each of the elements of a generative domain model can be implemented using different technologies, which gives rise to different *technology projections*:

– DSLs can be implemented as new textual languages (using traditional compiler building tools), embedded in a programming language (e.g., template metaprogramming in C++ or Template Haskell [17], OpenJava [18], OpenC++ [19], Metaborg [20]), graphical languages (e.g., UML profiles [21]), wizards and interactive GUIs (e.g., feature-based configurators such as Pure::Consul [22] or CaptainFeature [23]), or some combination of the previous. The appropriate structure of a DSL and the implementation technology depend on the range of variation that needs to be supported (see Figure 8). The spectrum ranges from routine configuration using wizards to programming using graphical or textual graph-like languages.
– Mappings can be realized using product configurators (e.g., Pure::Consul) or generators. The latter can be implemented using template and frame processors (e.g., TL [7], XVCL [24], or ANGIE [25]), transformation systems (e.g.,

DMS [26], StrategoXT [27], or TXL [28]), multi-staged programming [29], program specialization [30–32], or built-in metaprogramming capabilities of a language (e.g., template metaprogramming in C++ or Template Haskell).
– Components can be implemented using simply functions or classes, generic components (such as in the C++ STL), component models (e.g., JavaBeans, ActiveX, or CORBA), or aspect-oriented programming approaches (e.g., AspectJ [33], HyperJ [34], or Caesar [35]).

While some technologies cover all elements of a generative domain model in one piece (e.g., OpenJava or template metaprogramming in C++), a more flexible approach is to use an intermediate program representation to allow using different DSL renderings (e.g., textual or graphical) with different generator back-ends (e.g., TL or StrategoXT).

The choice of a specific technology depends on its technical suitability for a given problem domain and target users. For example, in the case of DSLs, concise textual languages may be best appropriate for expert users, but wizards may be better suited for novices and infrequent users. In the case of generator technologies, the need for complex, algebraic transformations may require using a transformation system instead of a template processor. Furthermore, there may be non-technical selection criteria such as mandated programming languages, existing infrastructure, familiarity of the developers with the technology, political and other considerations.

7 Model Driven Development and Generative Software Development

Perhaps the closest related area to generative software development is model-driven development (MDD), which aims at capturing every important aspect of a software system through appropriate models. A model is an abstract representation of a system (and possibly the portion of the world that interacts with it). A model allows answering questions about the system and its world portion that are of interest to the stakeholders of the system. They are better than the implementing code for answering these questions because they capture the intentions of the stakeholders more directly, are freer from accidental implementation details, and are more amenable to analysis. In MDD, models are not just auxiliary documentation artifacts; rather, models can be compiled directly into executable code that can be deployed at the customer's site.

There has been a trend in MDD towards representing models using appropriate DSLs, which makes MDD and generative software development closely related. Perhaps the main difference between MDD and generative software development is the focus of the latter on system families. While system families can be of interest to MDD, they are not regarded as a necessity.

Model-Driven Architecture (MDA) is a framework MDD proposed by the Object Management Group (OMG) [36]. While still being defined, the main goal of MDA is to allow developers to express applications independently of specific

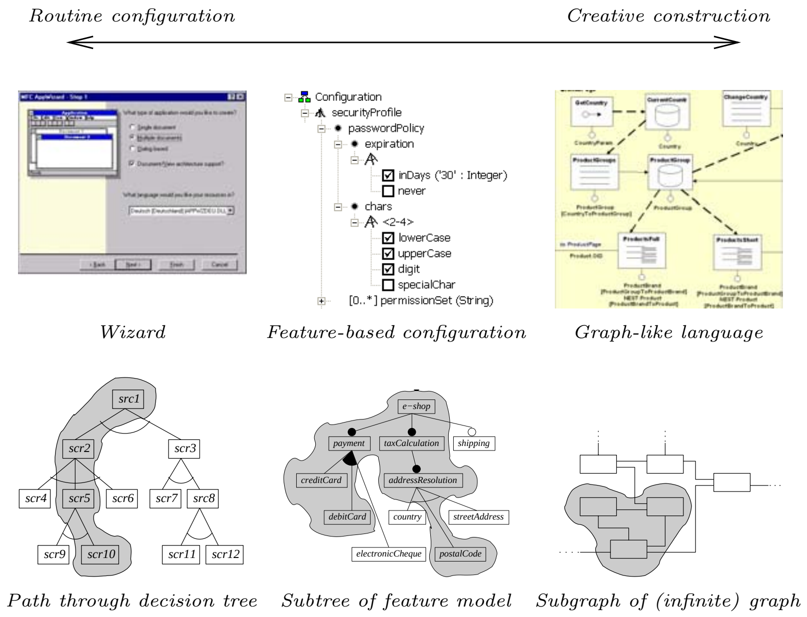


Fig. 8. Spectrum of DSL structures

implementation platforms (such as a given programming language or middleware). In MDA, an application is represented as a Platform Independent Model (PIM) that later gets successively transformed into series of Platform Specific Models (PSMs), finally arriving at the executable code for a given platform. The models are expressed using UML and the framework uses other related OMG standards such as MOF, CWM, XMI, etc. A standard for model transformations is work in progress in response to the Request for Proposals "MOF 2.0 Query/Views/Transformations" issued by OMG.

MDA concepts can be mapped directly onto concepts from generative software development: a mapping from PIM to PSM corresponds to a mapping from problem space to solution space. Beyond the similarities, there are interesting synergies. On the one hand, benefits of MDA include a set of standards for defining and manipulating modeling languages and the popularization of generative concepts in practice. Thanks to MDA, current UML modeling tools are likely to evolve towards low-cost DSL construction tools. On the other hand, the MDA efforts until now have been focusing on achieving platform independence, i.e., system families with respect to technology variation. However, generative software development addresses both technical and application-domain variability, and it may provide valuable contributions to MDA in this respect (see Figure 9). Often asked questions in the MDA context are (1) what UML profiles or DSLs

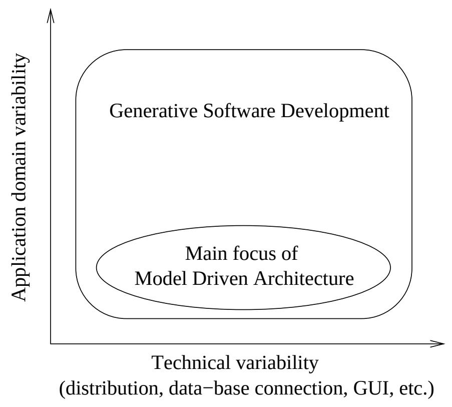

Generative Software Development
Main focus of
Model Driven Architecture
Application domain variability
Technical variability
(distribution, data−base connection, GUI, etc.)

Fig. 9. Relationship between generative software development and MDA

should be used to represent PIMs and (2) what is a platform in a given context. Domain analysis and domain scoping can help us to address these questions.

8 Other Related Fields

Figure 10 classifies a number of related fields by casting them against the elements of a generative domain model. Components, architectures, and generic programming are primarily related to the solution space. Aspect-oriented programming provides more powerful localization and encapsulation mechanisms than traditional component technologies. In particular, it allows us to replace many "little, scattered components" (such as those needed for logging or synchronization) and the configuration knowledge related to these components by well encapsulated aspectual modules. However, we still need to configure aspects and other components to implement abstract features such as performance properties. Therefore, aspect-oriented programming technologies such as AspectJ cover the solution space and only a part of the configuration knowledge. But aspects can also be found in the problem space, esp. in the context of DSLs used to described different aspects of a single system. Areas such as DSLs, feature modeling, and feature interactions address the problem space and the front part of the configuration knowledge. Finally, system-family and product-line engineering span across the entire generative domain model because they provide the overall structure of the development process (including domain and application engineering).

References

1. Weiss, D.M., Lai, C.T.R.: Software Product-Line Engineering: A Family-Based Software Development Process. Addison-Wesley (1999)

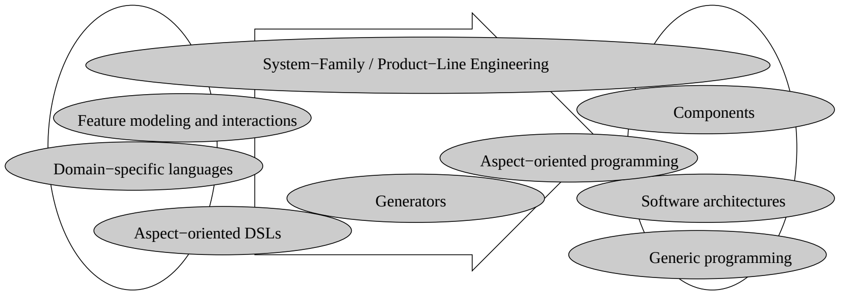

System−Family / Product−Line Engineering
Feature modeling and interactions
Domain−specific languages
Aspect−oriented DSLs
Generators
Aspect−oriented programming
Components
Software architectures
Generic programming

Fig. 10. Relationship between generative software development and other fields (from [37])

2. Clements, P., Northrop, L., eds.: Software Product Lines: Practices and Patterns. International Series in Computer Science. Addison-Wesley (2001)
3. Parnas, D.: On the design and development of program families. IEEE Transactions on Software Engineering **SE-2** (1976) 1–9
4. Neighbors, J.M.: Software Construction using Components. PhD thesis, Department of Information and Computer Science, University of California, Irvine (1980) Technical Report UCI-ICS-TR160. Available from `http://www.bayfronttechnologies.com/thesis.pdf`.
5. Cleaveland, J.C.: Building application generators. IEEE Software **9** (1988) 25–33
6. Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley (2000)
7. Cleaveland, C.: Program Generators with XML and Java. Prentice-Hall (2001)
8. Batory, D., Johnson, C., MacDonald, B., von Heeder, D.: Achieving extensibility through product-lines and domain-specific languages: A case study. ACM Transactions on Software Engineering and Methodology (TOSEM) **11** (2002) 191–214
9. Greenfield, J., Short, K.: Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. Wiley (2004) To be published.
10. Bosch, J.: Software product lines: Organizational alternatives. In: Proceedings of the 23rd International Conference on Software Engineering (ICSE). (2001)
11. Batory, D., O'Malley, S.: The design and implementation of hierarchical software systems with reusable components. ACM Transactions on Software Engineering and Methodology **1** (1992) 355–398
12. Kang, K., Cohen, S., Hess, J., Nowak, W., Peterson, S.: Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90TR -21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (1990)
13. Czarnecki, K., Helsen, S., Eisenecker, U.: Staged configuration using feature models. In Nord, R., ed.: Third Software Product-Line Conference, Springer-Verlag (2004)
14. DeBaud, J.M., Schmid, K.: A systematic approach to derive the scope of software product lines. In: Proceedings of the 21st International Conference on Software Engineering (ICSE), IEEE Computer Society Press (1999) 34–43
15. Buschmann, F., Jkel, C., Meunier, R., Rohnert, H., Stahl, M., eds.: Pattern-Oriented Software Architecture – A System of Patterns. International Series in Computer Science. John Wiley & Sons (1996)
16. Bosch, J.: Design and Use of Software Architecture: Adopting and evolving a product-line approach. Addison-Wesley (2000)

17. Czarnecki, K., O'Donnel, J., Striegnitz, J., Taha, W.: Dsl implementation in metaocaml, template haskell, and c++. [38] 50–71
18. M. Tatsubori: OpenJava: An extensible Java (2004) Available at `http://sourceforge.net/projects/openjava/`.
19. Sigeru Chiba: OpenC++ (2004) Available at `http://opencxx.sourceforge.net/index.shtml`.
20. Bravenboer, M., Visser, E.: Concrete syntax for objects. domain-specific language embedding and assimilation without restrictions. In C.Schmidt, D., ed.: Proceedings of the 19th ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'04). Vancouver, Canada. October 2004, ACM SIGPLAN (2004)
21. Jeff Grey et al.: OOPSLA'02 Workshop on Domain-Specific Visual Languages (2002) Online proceedings at `http://www.cis.uab.edu/info/OOPSLA-DSVL2/`.
22. pure-systems GmbH: Variant management with pure::consul. Technical White Paper. Available from `http://web.pure-systems.com` (2003)
23. Bednasch, T., Endler, C., Lang, M.: CaptainFeature (2002-2004) Tool available on SourceForge at `https://sourceforge.net/projects/captainfeature/`.
24. Wong, T., Jarzabek, S., Swe, S.M., Shen, R., Zhang, H.: Xml implementation of frame processor. In: Proceedings of the ACM Symposium on Software Reusability (SSR'01), Toronto, Canada, May 2001. (2001) 164–172 `http://fxvcl.sourceforge.net/`.
25. Delta Software Technology GmbH: ANGIE - A New Generator Engine (2004) Available at `http://www.delta-software-technology.com/GP/gptop.htm`.
26. Baxter, I., Pidgeon, P., Mehlich, M.: Dms: Program transformations for practical scalable software evolution. In: Proceedings of the International Conference on Software Engineering (ICSE'04), IEEE Press (2004)
27. Visser, E.: Program transformation with stratego/xt: Rules, strategies, tools, and systems. [38]
28. Cordy, J., Dean, T., Malton, A., Schneider, K.: Source transformation in software engineering using the txl transformation system. Information and Software Technology **44** (2002)
29. Taha, W.: A gentle introduction to multi-stage programming. [38]
30. Jones, N., Gomard, C., , Sestoft, P., eds.: Partial Evaluation and Automatic Program Generation. International Series in Computer Science. Prentice-Hall (1993)
31. Consel, C., Danvy, O.: Tutorial notes on partial evaluation. In: Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages, Charleston, SC, USA, ACM Press (1993) 493–501
32. Consel, C.: From a program family to a domain-specific language. [38] 19–29
33. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of aspectj. In: Proceedings of ECOOP'01. Lecture Notes in Computer Science, Springer-Verlag (2001)
34. Tarr, P., Ossher, H., Harrison, W., , Sutton, S.M.: N degrees of separation: Multi-dimensional separation of concerns. In: Proceedings International Conference on Software Engineering (ICSE) '99, ACM Press (1999) 107–119
35. Mezini, M., Ostermann, K.: Variability management with feature-oriented programming and aspects. In: Foundations of Software Engineering (FSE-12), ACM SIGSOFT (2004)
36. Object Management Group: Model-Driven Architecture (2004) `www.omg.com/mda`.
37. Barth, B., Butler, G., Czarnecki, K., Eisenecker, U.: Report on the ecoop'2001 workshop on generative programming. In: ECOOP 2001 Workshops, Panels and

Posters (Budapest, Hungary, June 18-22, 2001). Volume 2323 of Lecture Notes in Computer Science., Springer-Verlag (2001)

38. Christian Lengauer, D.B., Consel, C., Odersky, M., eds.: Domain-Specific Program Generation, International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003, Revised Papers. Volume 3016 of Lecture Notes in Computer Science. Springer-Verlag (2004)

Generative Programming from a Post Object-Oriented Programming Viewpoint

Shigeru Chiba

Dept. of Mathematical and Computing Sciences
Tokyo Institute of Technology
Email: `chiba@is.titech.ac.jp`

Abstract. This paper presents an application of generative programming to reduce the complications of the protocol for using an application framework written in an object-oriented language. It proposes that a programmable program translator could allow framework users to write a simple program, which is automatically translated by the translator into a program that fits the framework protocol. Then it mentions the author's experience with Javassist, which is a translator toolkit for Java, and discusses a research issue for applying this idea to real-world software development.

1 Introduction

Object-oriented programming languages have enabled us to develop component libraries that are often called *application frameworks*. A well-known simple example of such libraries is a graphical user interface (GUI) library. Since application frameworks provide a large portion of the functionality that application software has to implement, they can significantly reduce the development costs of application software.

However, application frameworks involve hidden costs. The developers who want to build their own application software with an application framework must first learn how to use the framework. Then they must write their programs to follow the protocol provided by the framework. These costs are considerably large if the framework provides relatively complex functionality. For example, to implement GUI with a typical GUI library (i.e. framework), the developers must learn the basic GUI architecture and a few concepts such as a callback and a listener. Then they must carefully write their programs to implement such a callback method and listener. To implement a web application on top of the J2EE framework, the developers must first take a tutorial course about J2EE programming and then write a program to follow the complicated J2EE protocol. For example, they must define two interfaces whenever they define one component class.

In this paper, we present an idea for reducing the hidden costs involved in application frameworks written in object-oriented languages. Our idea is to use a *programmable* program translator/generator, which automatically generates glue code for making the program written by a developer match the protocol supplied by an application framework. Thus the developer do not have to learn or follow the protocol given by the framework. Note that the program translator is not a fully-automated system. It is driven by a control program that is written by the framework developer. This is why the program translator used in our proposal is called programmable. In our idea, the framework must be supplied with the control program for customizing a program translator for that framework.

A research issue on this idea is how to design a language used to write a control program of the program translators/generator. We have developed a Java bytecode translator toolkit, named *Javassist* [1], and built several systems on top of that toolkit. Our experience in this study revealed that a programmable translator such as Javassist can be used to implement our idea. However, control programs for Javassist are still somewhat complicated and thus writing such a control program is not a simple task for framework developers. Studying a language for writing control programs is one of the future work.

2 Object-oriented Application Framework

Object-oriented programming languages enable a number of programming techniques, some of which are known as the design patterns [2]. These techniques play a crucial role in constructing a modern application framework. In some sense, they are always required to construct an application framework that provides complex functionality, in particular, non-functional concerns such as persistence, distribution, and user interface. The application framework that provides such functionality would be difficult to have simple API (Application Programming Interface) if object-oriented programming techniques are not used.

On the other hand, the users of such an application framework written in an object-oriented language must learn the protocol for using that framework. They must understand how design patterns have been applied to the framework, or they must know at least which methods should be overridden to obtain desirable effects and so on. These efforts are often major obstacles to use the application framework. A larger application framework tends to require a longer training period to the users of that framework.

The complications of such a framework protocol mainly come from the use of object-oriented programming techniques. For example, we below show a (pseudo) Java program written with the standard GUI framework. It is a program for showing a clock. If this program does not have GUI, then it would be something like the following simple and straightforward one (for clarifying the argument, the programs shown below are pseudo code):

```
class Clock {
  static void main(String[] args) {
    while (true) {
      System.out.println(currentTime());
      sleep(ONE_MINUTE);
    }
  }
}
```

This program only prints the current time on the console every one minute.

We can use the standard GUI library to extend this program to have better look. To do that, we must read some tutorial book of the GUI library and edit the program above to fit the protocol that the book tells us. First, we would find that the Clock class must extend Panel. Also, the Clock class must prepare a paint method for drawing a picture of clock on the screen. Thus you would define the paint method and modify the main method. The main method must call not the paint method but the repaint method, which the tutorial book tells us to call when the picture is updated. The following is the resulting program (again, it is pseudo code):

```
class Clock extends Panel {
  void paint(Graphics g) {
    // draw a clock on the screen.
  }
  static void main(String[] args) {
    Clock c = new Clock();
    while (true) {
      c.repaint();
      sleep(ONE_MINUTE);
    }
  }
}
```

Note that the structure of the program is far different from that of the original program. It is never simple or straightforward. For example, why do we have to define the paint method, which dedicates only to drawing a picture? Why does the main method have to call not the paint method but the repaint method, which indirectly calls the paint method? To answer these questions, we have to understand the underlying architecture of the framework provided by the GUI library. Since this architecture is built with a number of object-oriented programming techniques and most of tutorial books do not describe such details, understanding the underlying architecture is often difficult for "average" developers who do not have the background of GUI programming.

3 Protocol-less framework

To overcome the problem mentioned in the previous section, we propose an idea of using a *programmable* program translator. The users of an application framework should not be concerned about "the protocol" of a framework when writing their application programs. They should be able to write simple and intuitively understandable programs, which should be automatically translated into programs that fit the protocol for using the framework. I think that reducing the awareness about a framework protocol due to object-orientation is a key feature of post object-oriented programming.

Ideally, the transformation from the original Clock class into the GUI-based Clock class shown in the previous section should be performed automatically by a program translator instead of a human being. At least, the following modification for making the original program fit the protocol of the framework should be performed by a program translator:

- The Clock class must extend the Panel class. User classes of an application framework must often extend a class provided by the framework or implement an interface provided by the framework. Such class hierarchy should be automatically maintained by a program translator.
- The Clock class must declare the paint method. User classes of an application framework must often override some specific methods. Such overriding should be implicit. If necessary, the method drawing a picture should be able to have some other name than paint. If paint is not declared in user classes, the default method declaration of paint should be automatically added by a program translator.

Executing the automatic program transformation presented above is not realistic if any hints are not given. In our idea, this transformation is executed by a program translator controlled by a control program written by the developer of the application framework (Figure 1). Thus the program translators must be *programmable.* Since the framework developer knows the underlying architecture of that framework, writing such a control program should be fairly easy for her. Application frameworks should be distributed together with program translators and control programs of them.

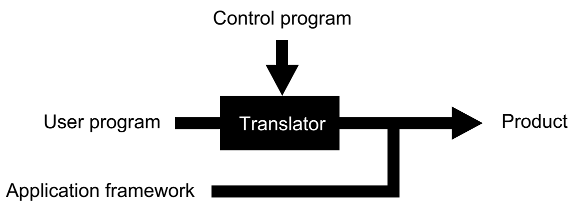


Fig. 1. Programmable program translator

4 Javassist

A challenge is to develop a good language for describing a control program given to the program translator in Figure 1. Toward this goal, we have been developing a Java bytecode translator toolkit named *Javassist* [1]. It is a Java class library for transforming a compiled Java program at the bytecode level (the bytecode is assembly code in Java).

A unique feature of Javassist is that it provides source-level abstraction for the developers who want to write a program for transforming Java bytecode.

There are several similar Java libraries that allow editing a class file (a compiled Java binary file). These libraries help the users read a class file, parse it, and produce objects that directly represent the internal data structures included in the class file. The users can modify the contents of the class file through these objects. However, since these objects directly correspond to the data structures in a class file, the users must learn the specifications of such internal data structures so that they can use these objects for modifying the contents of a class file. For example, they have to learn what the constant pool is and what the code attribute is. The former is a symbol table and the latter is a code block representing a method body.

Since Javassist provides source-level abstraction, the users of Javassist do not have to learn the specifications of the Java class file. Javassist translates the internal data structures in a class file into objects that represent the concepts familiar to Java developers (Figure 2). The users of Javassist can parse a class file and obtain objects representing a class, fields, methods, and constructors derived from the original class file. If the users change attributes of those objects, then the changes are reflected on the class file. For example, if the setName method is called on an object representing a class, Javassist changes the name of the class that the original class file represents.

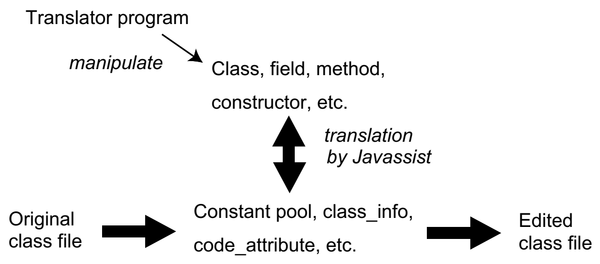


Fig. 2. Javassist translates bytecode-level concepts into source-level concepts

5 Translator programming language

Our experiences with Javassist for several years told us that developing a programmable program translator mentioned in Section 3 is a realistic idea. However, to actually use this idea for real-world software development, we need further study.

One of the open issues is a programming language for describing program transformation, that is, describing a control program in Figure 1. The source-level abstraction by Javassist has made it easier to write such a control program but making such a program sufficiently generic still needs further study. At least, one control program must be able to translate a number of user programs to fit the protocol of the application framework that the control program was written for. To do that, however, a control program must be able to recognize differences among user programs and find which parts of the code must be edited. For example, in the case of the Clock example shown above, the control program must find which class must extend the Panel class and which method is for drawing a picture on the screen. Maybe the users might have to give some hints to the control program but, if they must a large amount of hints, the resulting application framework would be as difficult to use as today's frameworks coming with a complicated protocol. To overcome this issue, other techniques studied in this research area — generative programming — should be introduced. For example, aspect-oriented programming and domain-specific language approaches might give some hints to solutions.

References

1. Chiba, S.: Load-time structural reflection in java. In: ECOOP 2000. LNCS 1850, Springer-Verlag (2000) 313–336
2. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison-Wesley (1994)

Generative Programming from an OOP/AOP Viewpoint

Pierre Cointe

article not received in time

Understanding Generative Programming
From (meta)objects to aspects

Pierre Cointe

OBASCO group, EMN-INRIA,
École des Mines de Nantes,
4 rue Alfred Kastler, La Chantrerie,
44307 Nantes Cedex 3, France
cointe@emn.fr

Abstract. Generative Programming (GP) is an attempt to manufacture sofware components in an automated way by developing programs that synthesize other programs. The purpose of this track is to introduce the *what* and the *how* of the GP approach from a programming language perspective. For the *what* we will learn lessons from object-oriented, component-based, aspect-oriented and domain specific languages. For the *how* we will discuss a variety of approaches and technics such as high-level compilation, meta-programming and metaobject protocols, programmable program translators , weavers, ...

1 Introduction

"The transition to automated manufacturing in software requires two steps. First, we need to move our focus from engineering single systems to engineering families of systems - this will allow us to come up with the "right" implementation components. Second, we need to automate the assembly of the implementation components using generators" [1].

Generative Programming (GP) is an attempt to provide a variety of approaches and techniques to manufacture sofware components in an automated way by developing programs that synthesize other programs [2]. According to the Czarnecki & Eisenecker book [1], the two main steps to industrialize software manufacturing are the modeling and the engineering of program family and the use of generators to automate the assembly of components. The purpose of this track is to have a better understanding of the *what and the* how of the GP approach from a programming language perspective. To achieve this goal we have invited the following speakers :

1. Krzysztof Czarnecki will give an *Overview of Generative Software Development* focusing on software reuse and development processes.

2. Pierre Cointe will present the *evolution of objects to metaobjects and aspects* from a reflective software engineering view point.

3. Shigeru Chiba will introduce *Generative Programming from a Post Object-Oriented Programming View point* by sketching an application for automating the use of an Object-Oriented framework.
4. Mira Mezini (and Klaus Ostermann) will discuss *Generative Programming from an AOP/CBSE Perspective* by introducing the AspectJ and Caeser languages. They will argue that general purpose programming languages should be augmented with abstraction mechanisms to encode and combine different crosscutting models instead of using program generation techniques to encode domain-specific knowledge.
5. Charles Consel will present *Generative Programming from a DSL Viewpoint* and will discuss how generative tools can be used to compile DSL programs into GPL programs.

2 Some lessons learnt from object-oriented languages

The object-oriented and reflective communities, together, have clearly illustrated the potential of separation of concerns in the fields of software engineering and open middleware [4, 7], leading to the development of aspect oriented programming [8, 6].

2.1 Reflection

The reflective approach makes the assumption that it is possible to separate in a given application, its *why* expressed at the base level, from its *how* expressed at the metalevel.

– In the case of a reflective object-oriented language *à la Smalltalk*, the principle is to reify at the metalevel its structural representation *e.g.,* its classes, their methods and the error-messages but also its computational behavior, *e.g.,* the message sending, the object allocation and the class inheritance. Depending on which part of the representation is accessed, reflection is said to be structural or behavioral. Meta-objects protocols (MOPs) are specific protocols describing at the meta-level the behavior of the reified entities. Specializing a given MOP by inheritance, is the standard way to extend the base language with new mechanisms such as multiple-inheritance, concurrency or metaclasses composition.
– In the case of an open middleware, the main usage of behavioral reflection is to control message sending by interposing a metaobject in charge of adding extra behaviors/services (such as transaction, caching, distribution) to its base object. Nevertheless, the introduction of such *interceptors/wrappers* metaobjects requires to instrument the base level with some *hooks* in charge of causally connecting the base object with its metaobject. Those metaobjects prefigured the introduction of AspectJ *crosscuts*, *e.g.,* the specification of execution points where extra actions should be woven in the base program [6, 9].

2.2 Separation Of Concerns

The Model-View-Controller *MVC* developed for Smalltalk is the first design-pattern making the notion of aspects explicit. The main idea was to separate, at the design level, the *model* itself describing the application as a class hierarchy and two separate concerns: the *display* and the *control*, themselves described as two other class hierarchies. At the implementation level, standard encapsulation and inheritance were not able to express these croscutting concerns and not able to provide the coupling between the model, its view, and its controller. This coupling necessitated:

- the introduction of a *dependence mechanism* in charge of notifying the observers when a source-object changes. This mechanism is required to automatically update the display when the state of the model changes.
- the instrumentation of some methods of the model to raise an event each time a given instance variable changes its value.

2.3 Aspects

On the one hand, object-oriented languages have demonstrated that *reflection* is a general conceptual framework to clearly modularize implementation concerns when the users fully understand the metalevel description. In that sense, reflection is solution oriented since it relies on the protocols of the language to build a solution. On the other hand, the *MVC* design-pattern has provided the developer with a problem-oriented methodology based on the expression and the combination of three separate concerns/aspects. The *MVC* was the precursor of *event programming* - in the Java sense - and contributed to the emergence of aspect-oriented programming [6, 5]. by making explicit the notion of *join-point*, *e.g.,* some well defined points in the execution of a *model* used to dynamically weave the aspects associated to the *view* and the *controller*.

3 Some open issues (to be developed later)

A first issue is to have a better understanding of how to use reflective tools to model aspects and their associated crosscutting languages and advice languages [3]. A second issue is to study the integration of aspects and objects to propose an alternative to inheritance as a mechanism for reuse. A third issue is to emphasize the use of reflection in the field of component adaptation as soon as self-reasoning is important. A fourth issue is to apply domain-specific languages to the expression of aspects.

References

1. Czarnecki, K., Eisenecker, U.W.: Generative Programming. Methods, Tools, and Applications. Addison-Wesley (2000).

2. Batory, D., Czarnecki, K., Eisenecker, U., Smaragdakis., Y., Sztipanivits J.: Generative Programming and Component Engineering. See http://www.cs.rice.edu/ taha/gpce/.
3. Tanter, E., Noyé, J., Caromel, D., Cointe, P,: Partial Behavioral Reflection: Spatial and Temporal Selection of Reification. Proceedings of the 18th ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications. OOPSLA 2003. ACM SIGPLAN Notices, volume 38, number 11, pages 27-46.
4. Thomas, D.: Reflective Software Engineering - From MOPS to AOSD. Journal Of Object Technology, volume 1, number 4, pages 17-26. October 2002.
5. Wand, M.: Understanding Aspects. Invited talk at ICFP 2003. Available at www.ccs.neu.edu/home/wand/ICFP 2003
6. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C, Loingtier, J.-M., Irwin, J.: Aspect-Oriented Programming. ECOOP'97 - Object-Oriented Programming - 11th European Conference, volume 1241, pages 220–242.
7. Cointe, P.: Les langages à objets. Technique et Science Informatique (TSI), volume 19, number 1-2-3, (2000).
8. Aksit, M., Black, A., Cardelli, L., Cointe. P., Guerraoui, R. (editor), and al,: Strategic Research Directions in Object Oriented Programming, ACM Computing Surveys, volume 8, number 4, page 691-700, (1996).
9. . Douence, R., Motelet, O., Südholt, M.: A formal definition of crosscuts. Proceedings of the 3rd International Conference on Reflection 2001, LNCS volume 2192, pages 170-186, (2001).

Author Index

With the additional support of

INRIA · UNIVERSITÉ D'EVRY VAL D'ESSONNE · UNIVERSITE DE RENNES 1 · Microsoft Research · IRISA institut de recherche en informatique et systèmes aléatoires · genopole · CNRS CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE

information
society
technologies

FET - Future and
Emerging Technologies

DIMACS — Center for
Discrete Mathematics
& Theoretical
Computer Science

European Research Consortium
for Informatics and Mathematics
ERCIM
www.ercim.org

This workshop is part of a series of strategic workshops to identify key research challenges and opportunities in Information Technology. These workshops are organised by ERCIM, the European Research Consortium for Informatics and Mathematics, and DIMACS the Center for Discrete Mathematics & Theoretical Computer Science. This initiative is supported jointly by the European Commission's Information Society Technologies Programme, Future and Emerging Technologies Activity, and the US National Science Foundation, Directorate for Computer and Information Science and Engineering.

More information about this initiative, other workshops, as well as an electronic version of this report are available on the ERCIM website at http://www.ercim.org/EU-NSF/