The Chemical Machine: An Interpreter for the Higher Order Chemical Language

Vilmos Rajcsányi and Zsolt Németh

MTA SZTAKI Computer and Automation Research Institute,
P.O. Box 63, H-1518 - Hungary
zsnemeth@sztaki.hu

Abstract. The notion of chemical computing has evolved for more than two decades. From the seminal idea several models, calculi and languages have been developed and there are various proposals for applying chemical models in distributed problem solving where some sort of autonomy, self-evolving nature and adaptation is sought. While there are some experimental chemical implementations, most of these proposals remained at the paper-and-pencil stage. This paper presents a general purpose interpreter for the Higher Order Chemical Language. The design follows that of logic/functional languages and bridges the gap between the highly abstract chemical model and the physical machine by an abstract interpreter engine. As a novel approach the engine is based on a modified hierarchical production system and turns away from imperative languages.

1 Introduction

The advent of large scale distributed systems (such as grids, service oriented architectures) introduced a group of problems that are hard to solve by humans or by any machinery in an exact way due to the very large number of entities, their heterogeneous nature, partial lack of information of their state, unpredictable, error prone behavior and many other factors. Approximately the same time appeared the notion of autonomic computing [17] where entities are supposed to monitor and control themselves according to some strategies: self-configuration, self-optimization, self-healing and self-protection. Since then a large number of reflective, self-* properties of computing entities have been proposed and realized. This new notion of computing naturally attracted non-conventional approaches; in fact the seminal paper [17] also took inspiration from the nervous system [13]. There is a large group of models that mimic various biological, chemical, physical, ethological processes and phenomenons or simply take them as metaphors.

In the chemical programming paradigm, instead of computing steps (instructions) and their strict order, a program is conceived as a chemical solution where data and procedures are molecules floating around and computation is a series of reactions between these molecules. Note, that in this case chemistry is just an inspiration or an abstract metaphor as opposed to chemical models (artificial chemistries) where computation closely simulates some chemical processes [12]. This vision of chemical computing is formalized in the γ-calculus [3] as (without the chemical guise) a declarative functional computational model where terms are commutative and associative.

M. Alexander et al. (Eds.): Euro-Par 2011 Workshops, Part I, LNCS 7155, pp. 181–190, 2012.

The chemical model and the γ-family (the calculus and the related languages) has already been investigated in various distributed scenarios, like self-organizing systems [8] where a self-healing, self-optimizing and self-protecting mail system is studied. Grids are obviously a good target for applying the chemical model in some well-known problems like coordinating a ray-tracing example on desktop grids [7], enacting workflows on-the-fly with strong emphasis on dynamicity both in the environment and in the workflow structure [11] and modeling self-developing secure virtual organisations [2]. Recently service oriented techniques and clouds also attracted great attention and proposals like chemical based service orchestration [6], dynamic service composition [5], dynamic service composition with partial instantiations and re-using instantiations [18] and others. Note, that the chemical model in all these cases is not applied for problem solving (in terms of solving any computational tasks) but *coordinates* the execution so that it may exhibit some of the features of the chemical metaphor like timely response to events, adaptation, self-evolution, intrinsic concurrency, independency, maximum parallelism and many others.

Albeit application of the chemical metaphor in grids and service oriented systems is well studied and various concepts are elaborated, appropriate interpreter and development tools for executing programs expressed in the chemical metaphor are largely missing. Most of these models require framework that is (i) able to execute the code expressed in a chemical language and (ii) provides interfaces to the embedding system so that some processes can be controlled by the chemical program meanwhile monitored data can be gathered. The work introduced in this paper is focused on (i) and aimed at creating an *interpreter* that supports the entire Higher Order Chemical Language (HOCL), a language that is based on and extends the γ-calculus. While the chemical model is quite different from any other widespread computing models and languages, careful study revealed similarities in other paradigms and the combination of techniques related to declarative languages and those of production systems allowed a realization of the interpreter in a short development cycle. At deciding the implementation means attention was paid to (ii) so that the interpreter can be interfaced with various tools and environments in the future. The work is focusing on the *design* and *realization* of the interpreter. Establishing autonomic or adaptive behaviour in the chemical framework is on one hand presented in papers [7][11] [5][8][18], etc., on the other hand related to the *application* of the interpreter and not presented here.

2 The Chemical Computational Model

Most algorithms are expressed sequentially even if they describe inherently parallel activities. Gamma (General Abstract Model for Multiset Manipulation) [4] aimed at relaxing the artificial sequentializing of algorithms. It is a multiset rewriting system where the program is represented by a set of declarative rules that are atomic, fire independently and potentially simultaneously, according to *local* and *actual* conditions. There is no concept of any centralized control, ordering, serialization rather, the computation is carried out in a non-deterministic, self-evolving way. It has been shown

in [4] that some fundamental problems of computer science (sorting, prime testing, string processing, graph algorithms, etc.) can be expressed in Gamma in a concise and elegant way.

The γ-calculus is a formal definition of the chemical paradigm. The fundamental data structure is the multiset M. γ-terms (molecules) are: variables x, γ-abstractions $\gamma\langle x\rangle.M$, multisets (M_1, M_2) and solutions $\langle M\rangle$. Juxtaposition of γ-terms is commutative $(M_1, M_2 \equiv M_2, M_1)$ and associative $(M_1, (M_2, M_3) \equiv (M_1, M_2), M_3)$. Commutativity and associativity are the properties that realize the 'Brownian-motion', i.e., the free distribution and unspecified reaction order among molecules . The γ-abstractions are the reactive molecules that can take other molecules or solutions and replace them. Due to the commutative and associative rules, the order of parameters is indifferent; molecules, solutions participating in the reaction are extracted by pattern matching – *any* of the matching ones may react. The semantics of a γ-reduction is $(\gamma\langle x\rangle.M), \langle N\rangle \rightarrow_\gamma M[x := N]$ i.e., the two reacting terms on the left hand side are replaced by the body of the γ-abstraction where each free occurrence of variable x is replaced by parameter N if N is inert . Reactions may depend on certain conditions expressed as C in $\gamma\langle x\rangle\lfloor C\rfloor.M$ that can be reduced only if C evaluates to true before the reaction . Reactions can capture multiple molecules in a single atomic step. The universal symbol ω matches any pattern. Reactions are governed by: (i) law of locality , i.e. if a reaction can occur, it will occur in the same way irrespectively to the environment; and (ii) membrane law , i.e. reactions can occur in nested solutions or in other words, solutions may contain sub-solutions separated by a membrane. The γ-calculus is a *higher order* model, where abstractions – just like any other molecules – can be passed as parameters or yielded as a result of a reduction [8][3].

The Higher Order Chemical Language (HOCL) [3] is a language based on the Gamma principles more precisely, the γ-calculus extended with expressions, types, pairs, empty solutions and names. HOCL uses the self-explanatory **replace... by... if...** construct to express rules. **replace** P **by** M **if** C formally corresponds to $\gamma(P)\lfloor C\rfloor.M$ with a major difference: while γ-abstractions are destroyed by the reactions, HOCL rules are n-shot and remain in the solution nevertheless, single-shot γ-style rules can also be added. **replace... by... if...** is followed by **in** $\langle ...\rangle$ that specifies the solution the active molecule floats in. Notable features (extensions) of HOCL are: types,= that can be added to patterns for matching; pairs in form of $A_1 : A_2$ where A_1 and A_2 are atoms; and naming that allows to identify and hence, match rules, e.g. $\mathbf{let}\, inc = \mathbf{replace}\, x\, \mathbf{by}\, x+1\, \mathbf{in}\, \langle 1, 2, 3, inc\rangle$ specifies an active molecule called inc which captures an integer and replaces it with its successor, floating in a solution together with integers 1, 2, 3. Some possible reduction steps can be (note, the model is non-deterministic, there are different possible execution paths):

$$\langle 1, 2, 3, inc\rangle \rightarrow \langle 2, 2, 3, inc\rangle \rightarrow \langle 3, 2, 3, inc\rangle \rightarrow \langle 3, 2, 4, inc\rangle \rightarrow \langle 3, 3, 4, inc\rangle$$

3 The Concept of the Chemical Interpreter

In case of declarative languages, the semantics of the execution model and that of the underlying physical architecture is quite different therefore, they are usually executed

via an abstract, hypothetic engine placed inbetween. The program is first transformed (compiled) into the language of the abstract engine that successively interprets the input and executes it. From the programmer's point of view the abstract engine is a machine that is able to execute the high-level language natively, it hides all the details of the real physical machine whereas, the abstract engine and its language is closer to the physical machine and can be executed in a simpler way (the semantic gap is narrower.) The most known such engine is the Warren's Abstract Machine (WAM) for executing Prolog [1] or SECD and Lispkit [15] for executing functional languages but there are many such examples like some implementations of (early) Pascal [19] or less known and more specific languages like Palingol [10].

The design of our chemical engine is also based on this principle. Thus, in our approach HOCL is first transformed into the code of the abstract engine and then this intermadiate code is interpreted. It is easy to see that HOCL execution resembles that of (i) functional languages with the exception of commutative and associative properties and (ii) production systems with the exception of hierarchical knowledge base and concurrent execution; yet not equivalent to any of these. To shorten the development cycle we carefully examined the similarities and differences in the computational models and opted to realize the HOCL abstract engine based on the notion of a production system. A production system consists of facts (knowledge) and rules (behaviour) applied to facts. If the facts fulfill conditions assigned to a certain rule, the rule is activated. From many activated rules one is selected by conflict resolution and fired. Firing a rule means executing its action part that updates the facts and leads to firing further rules. This so called production cycle is repeated over again.

Some of the key requirements of an efficient and simple realization of interpreting HOCL. (i) Efficient pattern matching. Production systems often apply the RETE-algorithm [14] in such a way, a highly efficient pattern matching, the most important cornerstone of the realization is available ready-made. This is the main inspiration of realizing the HOCL interpreter on the foundation of a production system. (ii) Nested solutions (hierarchical knowledge base). Most production systems assume a global knowledge base and do not allow the structured or hierarchical facts. This aspect needs a careful elaboration in the HOCL abstract engine as it is different in production systems. (iii) Concurrency. The concept of locality (molecules react with their "neighbor" molecules) is simulated by a random choice of potential molecules. Yet, the dynamics of the chemical system is quite different from that of a production system and the random conflict resolution needs further refinement. (iv) Level of parallelism. The γ-model is inherently concurrent and this behavior should be modeled with multiple concurrent execution threads yet, their level (granularity) can be different. To keep the granularity at a reasonable level yet, to enable concurrent behavior, we assigned an execution thread to each solution thus, solutions can evolve independently whereas concurrency within a solution (race condition among molecules) is represented by random choice of reacting molecules.

The conceptual representation of various elements of a HOCL program will be introduced by Dijkstra's Dutch flag [4], as an example. The aim of the Dutch flag problem is to order three colors, white, red and blue in a randomized array so that they are arranged according to the stripes of the Dutch national flag: red, white and blue.

$$\begin{aligned}
&\textbf{let } red = \textbf{replace } \langle i, red\rangle, \langle j, white\rangle \textbf{ by } \langle i, white\rangle, \langle j, red\rangle \textbf{ if } i > j \textbf{ in}\\
&\textbf{let } white = \textbf{replace } \langle i, white\rangle, \langle j, blue\rangle \textbf{ by } \langle i, blue\rangle, \langle j, white\rangle \textbf{ if } i > j \textbf{ in}\\
&\textbf{let } blue = \textbf{replace } \langle i, red\rangle, \langle j, blue\rangle \textbf{ by } \langle i, blue\rangle, \langle j, red\rangle \textbf{ if } i > j \textbf{ in}\\
&\langle\langle 1, blue\rangle, \langle 2, white\rangle, \langle 3, white\rangle, \langle 4, red\rangle, \langle 5, blue\rangle, \langle 6, white\rangle, red, white, blue\rangle
\end{aligned}$$

We introduce a simplified, easy-to-read pseudo code for representing and explaining the code of the production system. While they show all the necessary information many irrelevant details are eliminated. A production system represents its knowledge in facts like (1) or (1 2 3). Some facts can have named slots like ((x 1) (y 2) (z 3)). A rule has a left hand side (LHS) pattern that must be matched to enable the rule followed by $\Rightarrow$ and a right hand side (RHS) action that is triggered if the rule fires.

Molecules. As one may expect, a *passive molecule* is simply transformed into a fact like $1 \rightarrow$ `(molecule (value 1))` or $red \rightarrow$ `(molecule (color red))`. A straightforward (and naive) approach would be to represent *active molecules* as production rules. This way however, makes it very hard to realize the higher order property of the HOCL model where active molecules can be captured transformed, canceled or added just like any other molecule. Therefore, active molecules are represented by a rule *and* a fact. Thus, molecule red is transformed into a fact `(rule red)` and a rule with pattern shown as (some parts to be refined later):

```
(defrule red
         (rule red)
         ;match <i, red> and <j, white> if i>j
       ⇒
         ;swap <i, red> and <j, white>
```

This rule can fire if fact `(rule red)` is present in the same solution. All modifications to the active molecules (added, withdrawn, transferred) are performed on this fact that enables the rule. For instance, moving the active red molecule from one solution to another is simply moving the `(rule red)` fact.

Solutions are two faced entities: they are data if inert and are separate running processes (and thus, unable to be matched) if active. Solutions can hold passive molecules, active molecules, other solutions or pairs and can be nested in arbitrary depth. Unfortunately, production systems usually do not allow nesting the facts hence, there is no straightforward representation. We opted for a Prolog-like representation of compound terms [1] where not actual terms but references to terms that are stored. Therefore, molecules are augmented with identifiers so that references can be put to them. For instance, $\langle 1, blue\rangle$ is represented as two facts `(molecule (value 1) (in` id_k`))` and `(molecule (color blue) (in` id_k`))` and then the solution itself is a fact `(solution` id_k`)` (just the idea is shown here, there is more information related to solutions and molecules). This representation seemingly calls for a complicated recursive pattern-matching but it can be solved very efficiently in a flat manner as (following the above example):

```
(defrule red
         (rule red)
         (solution x)
         (molecule (value i)(in x))
         (molecule (color red)(in x))
         (solution y)
         (molecule (value j)(in y))
         (molecule (color white)(in y))
         (test i > j)
       ⇒
         ;swap <i, red> and <j, white>
```

where the matching variables represent the constraint so that molecules belonging to the given solution are selected. Similarly, multiply nested solutions are represented in the same way. Pairs are a special case of solutions: they have exactly two molecules inside and their order *is* relevant. With minor differences, all the principles introduced for solutions are used for pairs, too.

Transfer between Solutions. Molecules can be moved between solutions for instance, in **replace** $\langle i, red\rangle, \langle j, white\rangle$ **by** $\langle i, white\rangle, \langle j, red\rangle$ the two color molecules are exchanged between the two solutions. This is a very simple example but there are cases where multiple molecules are moved, or every molecule (ω) moved except some. Furthermore, deleting a molecule can be traced back to the same situation where it is taken from a solution but put nowhere. In order to handle all these cases efficiently and uniformly, we categorized the following cases as types

- **replace** $a : \langle \omega_a\rangle, b : \langle \omega_b\rangle$ **by** $a : \langle\rangle, b : \langle \omega_a, \omega_b\rangle$ – moving *all* molecules, e.g. from solution tagged a to solution b
- **replace** $a : \langle a, b, c, \omega_a\rangle, b : \langle \omega_b\rangle$ **by** $a : \langle \omega_a\rangle, b : \langle a, b, c, \omega_b\rangle$ – moving certain molecules, e.g. a, b, c form solution a to solution b
- **replace** $a : \langle a, b, c, \omega_a\rangle, b : \langle \omega_b\rangle$ **by** $a : \langle a, b, c\rangle, b : \langle \omega_a, \omega_b\rangle$ – moving all but certain molecules, e.g. all molecules from solution a to b except a, b, c

They can be further classified if the source and target solutions are top-level or nested ones or nil. Altogether 15 types of operations belong to this category. In fact, in reactions most of the actions are putting molecules around therefore, this operation must be very simple in the language (and efficient in the implementation). The intermediate language therefore is extended with `(relocate toMove, notToMove, from, to)`, a special custom function. Thus, we can finalize the example as

```
(defrule red
 (rule red)
 (solution x)
 (molecule (value i)(in x))
 (molecule (color red)(in x))
 (solution y)
 (molecule (value j)(in y))
 (molecule (color white)(in y))
 (test i > j)
⇒
```

```
(relocate (molecule (value i)(in x)) nil (solution x)(solution y))
(relocate (molecule (value j)(in y)) nil (solution y)(solution x))
```

Hence, the active molecule red has been rewritten into `rule red` of the intermediate language. It is important to mention that – just like the HOCL reaction – firing a rule is an atomic step. That is, in the above example molecules are transferred in a single step and there are no intermediate inconsistent states.

4 Implementation

The principles of an HOCL interpreter based on a production system drafted above have been implemented in jess [16], a Java based production system. Here some additional, implementation related details are explained only.

The Intermediate Language. HOCL programs are transformed (compiled) into an intermediate language that is based on the jess script language with (i) some restrictions and (ii) an added function. Restriction means a fixed template of molecules and a strict pattern in the head of rules. These principles were shown in Section 3 but in reality molecules contain more information (technical details) than presented before; there is an inherent need to keep them consistent. Therefore, there is a `molecule` template that defines all the necessary slots and all other passive molecules are derived from that. Restrictions are also present in the head of rules: capturing a molecule has a certain pattern sequence that must be strictly followed. The added function is the `relocate` introduced earlier. It is important to notice that this is the only one function that is not part of the jess script language and a large area of possible cases are realized by this single instruction. Due to the minimal changes introduced, the intermediate language is very close to the jess script. One familiar with jess or other production systems can easily read, understand and modify the intermediate language. Minimal changes also ensure that the intermediate language is executed as efficiently as the native jess script.

The Interpreter. We kept the same principle: introduce as little changes as possible thus, the HOCL interpreter is just a slightly modified jess engine. Furthermore, in case of the interpreter all these changes are transparent to the user. Albeit invisible, some important modifications and extensions must have been added to the basic execution mechanism of jess mainly due to the required support of hierarchical knowledge base. These include activities related to spawning a new RETE-engine (initiate a new solution) or opposedly, stop a RETE-engine. In such cases transferring data to a new process and vice versa by maintaining consistency, correctness and avoid synchronization problem is a complex task. To achieve these goals efficiently, Java procedures operate on the internal data structures of jess. Similarly, the realization of *relocate* is encoded in the interpreter as a custom Java function. Furthermore, the random conflict resolution must have been modified, see the explanation in Section 5. Measurements showed that these additional functionalities in the jess engine do not add significant overhead or cause performance degradation.

User Interface, Program Control and External Interfaces. There is a simple user interface developed that facilitates the execution, tracing and debugging of HOCL programs (Figure 1). The main fields show the current reactions, the possible reactions in each solution and the solution structure. The latter is augmented with some graphical aids to see where reactions are possible and what are the inert solutions. Solutions and molecules are clickable: new molecules can be added to solutions at run-time whereas breakpoints can be added or withdrawn on active molecules. Tracing and debugging is supported by various run modes: step-by-step, continuous run with variable speed and breakpoint. For specific applications custom-made user interfaces can be made for instance, an experimental tic-tac-toe game table was implemented (see Section 5). This latter also demonstrates how easily the interpreter can be interfaced with other programs that is a fundamental requirement for coordinating tasks the chemical paradigm is aimed at. In this case the game board is a separate process and steps made by the user are external events imported into the chemical engine whereas steps made by the computer are events that are exported and displayed graphically. In the same way, other sources of events and control can be realized in different scenarios.

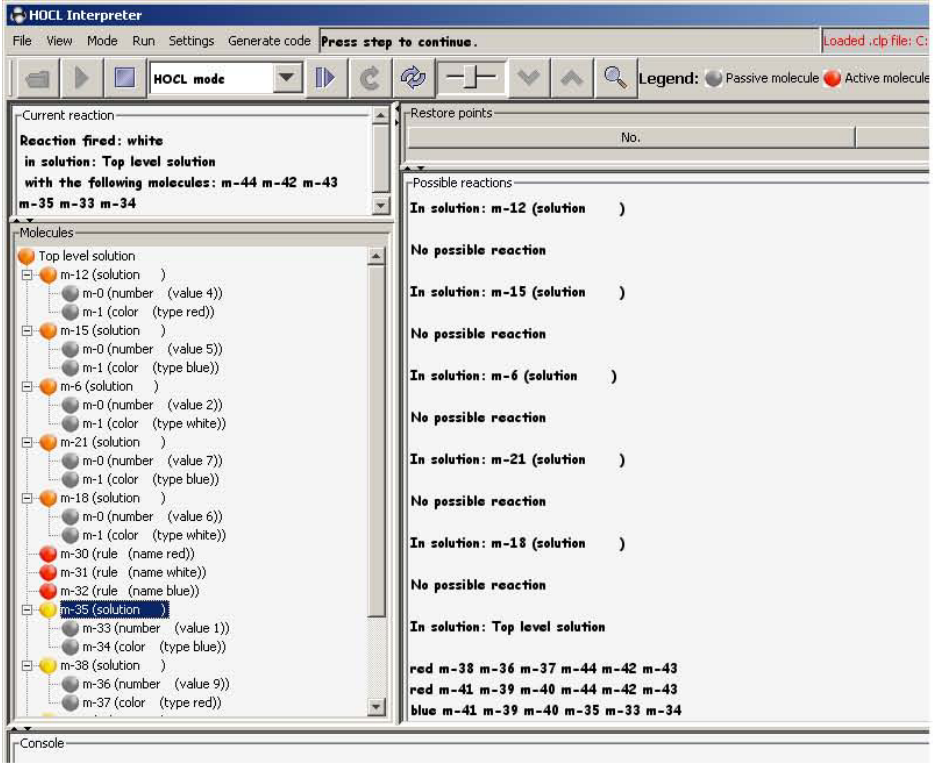

Fig. 1. Graphical interface for the HOCL interpreter

5 Experiences Learned

The interpreter was tested by a large set of toy examples to verify the correctness of elementary constructs in the language. Also, it was tested by some nontrivial problems listed in [4][8]. Here we present two experiences we learned beyond the simple correctness tests.

An implementation of the foxes and rabbits problem (Lotka-Volterra equations [9]) revealed that dynamicity in the chemical model is a crucial issue. This type of applications should oscillate (the number of foxes and rabbits change periodically) but our initial attempts diverged. The problem was caused by the random conflict resolution of the production system that did not really simulate the random mixture of molecules and must have been replaced by a custom made one. While this sensitivity seemingly affects a very little portion of computational problems, the chemical approach is associated with realizing self-* autonomic systems where evolution and dynamicity of certain populations is of fundamental importance and such aspects must be carefully researched and elaborated.

A player vs. machine tic-tac-toe game revealed the importance of the appropriate transformation of HOCL into the intermediate language. A very simple implementation of this game was encoded in HOCL in a concise way and was executed by the interpreter. Yet, as the size of the field grew, performance problems started to appear and around the table size of 30*30 the game became unplayable due to large response times. The root of the problem was in expressing the HOCL program in the intermediate language. While HOCL allows a very expressive and elegant problem statement, pattern matching works more efficiently on numerous but simple rules. Therefore, a complex HOCL statement must be transformed into the intermediate language so that it is broken into simpler, more specific rules that facilitate pattern matching; tic-tac-toe was successfully hand coded so that it became scalable. While the transformation of HOCL into the intermediate language (compiler construction) can be described easily, taking into consideration such performance issues requires more research work.

6 Conclusions

In this paper we presented the design and implementation principles of an HOCL interpreter for executing programs written in a higher order chemical language. The chemical computing model is an upcoming candidate for realizing autonomic properties in various distributed settings (such as grid and service based environments, see [5] [11] [18]).

The proposed execution of HOCL programs is an interpreter realized as an abstract engine. The engine is based on a production system that lends its state-of-the-art pattern matching mechanism but modified to support the hierarchical notion of knowledge base of the chemical semantics and fulfill other technical challenges. The interpreter supports the entire HOCL language and has a graphical user interface and a basic support for tracing and debugging. The realization of the interpreter also makes possible to interface it to other systems for observation and control.

Test experiments proved the correctness of the interpreter. They also revealed the importance of efficient representation of the HOCL program at the intermediate level

and that of the dynamic behavior. Both are strongly related to self-evolving properties of autonomic systems and therefore, will play crucial role in real-life applications. These aspects are targets of further research.

Acknowledgements. The research leading to these results has received funding from the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement 215483 (S-Cube).

References

1. Aït-Kaci, H.: Warren's Abstract Machine: A Tutorial Reconstruction. MIT Press (1991)
2. Arenas, A.E., Banâtre, J.-P., Priol, T.: Developing Autonomic and Secure Virtual Organisations with Chemical Programming. In: Guerraoui, R., Petit, F. (eds.) SSS 2009. LNCS, vol. 5873, pp. 75–89. Springer, Heidelberg (2009)
3. Banâtre, J.-P., Fradet, P., Radenac, Y.: Generalised multisets for chemical programming. Math. Struct. in Comp. Science 16, 557–580 (2006)
4. Banâtre, J.-P., Le Métayer, D.: Programming by multiset transformation. Commun. ACM 36(1), 98–111 (1993)
5. Banâtre, J.-P., Priol, T.: Chemical programming of future service-oriented architectures. JSW 4(7), 738–746 (2009)
6. Banâtre, J.-P., Priol, T., Radenac, Y.: Service Orchestration Using the Chemical Metaphor. In: Brinkschulte, U., Givargis, T., Russo, S. (eds.) SEUS 2008. LNCS, vol. 5287, pp. 79–89. Springer, Heidelberg (2008)
7. Banâtre, J.-P., Le Scouarnec, N., Priol, T., Radenac, Y.: Towards "chemical" desktop grids. In: eScience, pp. 135–142 (2007)
8. Banâtre, J.-P., Fradet, P., Radenac, Y.: Programming self-organizing systems with the higher-order chemical language. International Journal of Unconventional Computing 3(3), 161–177 (2007)
9. Berryman, A.A.: The origins and evolution of predator-prey theory. Ecology (73) (1992)
10. Billoud, B., Kontic, M., Viari, A.: Palingol: a declarative programming language to describe nucleic acids secondary structures and to scan sequence database. Nucleic Acids Res. (24), 1395–1403 (1996)
11. Caeiro, M., Németh, Z., Priol, T.: A chemical model for dynamic workflow coordination. In: PDP, pp. 215–222 (2011)
12. Dittrich, P., Ziegler, J., Banzhaf, W.: Artificial chemistries-a review. Artificial Life 7(3), 225–275 (2001)
13. Dobson, S., Sterritt, R., Nixon, P., Hinchey, M.: Fulfilling the vision of autonomic computing. IEEE Computer 43(1), 35–41 (2010)
14. Forgy, C.: Rete: A fast algorithm for the many patterns/many objects match problem. Artif. Intell. 19(1), 17–37 (1982)
15. Henderson, P.: Functional programming - application and implementation. Prentice Hall International Series in Computer Science, pp. 1–355. Prentice Hall (1980)
16. Hill, E.F.: Jess in Action: Java Rule-Based Systems. Manning Publications Co., Greenwich (2003)
17. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. IEEE Computer 36(1), 41–50 (2003)
18. Di Napoli, C., Giordano, M., Pazat, J.-L., Wang, C.: A Chemical Based Middleware for Workflow Instantiation and Execution. In: Di Nitto, E., Yahyapour, R. (eds.) ServiceWave 2010. LNCS, vol. 6481, pp. 100–111. Springer, Heidelberg (2010)
19. Nori, K.V., Ammann, U., Jensen, K., Nageli, H.H., Jacobi, C.: Pascal-p implementation notes. In: Pascal - The Language and its Implementation, pp. 125–170 (1981)