

Deployment and Evaluation of a Decentralised Runtime for Concurrent Rule-based Programming Models

Marko Obrovac, Cédric Tedeschi

▶ **To cite this version:**

Marko Obrovac, Cédric Tedeschi. Deployment and Evaluation of a Decentralised Runtime for Concurrent Rule-based Programming Models. [Research Report] RR-8145, INRIA. 2012, pp.20. hal-00755997

HAL Id: hal-00755997
https://inria.hal.science/hal-00755997

Submitted on 22 Nov 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

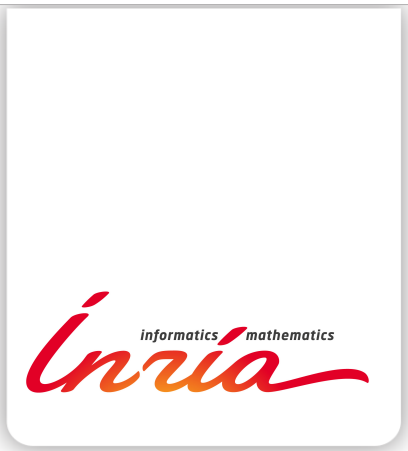


Deployment and Evaluation of a Decentralised Runtime for Concurrent Rule-based Programming Models

Marko Obrovac, Cédric Tedeschi

RESEARCH REPORT
N° 8145
November 2012
Project-Teams Myriads

ISSN 0249-6399 ISRN INRIA/RR--8145--FR+ENG



Deployment and Evaluation of a Decentralised Runtime for Concurrent Rule-based Programming Models

Marko Obrovac, Cédric Tedeschi

Project-Teams Myriads

Research Report n° 8145 — November 2012 — 20 pages

Abstract:
With the emergence of highly heterogeneous, dynamic and large distributed platforms, declarative programming, whose goal is to ease the programmer's task by separating the control from the logic of a computation, has regained a lot of interest recently, as a means of *programming* such platforms.
In particular, rule-based programming, which allows to simply specify crucial features such as communication protocols or computing workflows, is regarded as a promising model in this quest for adequate programming abstractions for these platforms. However, while these models are gaining a lot of attention, there is a demand for generic tools able to run such models at large scale.
The chemical programming model, which was designed following the chemical metaphor, is a rule-based programming model, with a non-deterministic execution specification, where rules are applied concurrently, on a multiset of objects. In this paper, we explore the experimental side of concurrent rule-based models, by deploying a distributed chemical runtime at large scale.
The architecture proposed combines a peer-to-peer communication layer with an adaptive protocol to atomically capture objects on which rules should be applied, and an efficient termination detection scheme. We describe the software prototype fully implementing this architecture. Based on its deployment over a large real-world test-bed, we present its performance results, which confirm analytically obtained complexities, and experimentally show the sustainability of such a programming model.

Key-words: Rule-based programming models, chemical computing, deployment, large-scale experiments

RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Déploiement et évaluation d'un environnement d'exécution décentralisé pour les modèles de programmation à base de règles

Résumé :

Avec l'émergence de plates-formes réparties à large échelle, et hautement hétérogènes et dynamique, la programmation déclarative, dont le but est de faciliter la tâche du programmeur en séparant le contrôle d'un calcul de sa logique, a été récemment désigné comme un moyen de *programmer* de telles plates-formes.

En particulier, les langages à base de règles, qui permettent de spécifier de façon simple des éléments cruciaux des systèmes distribués (comme les protocoles de communications) semble un modèle prometteur dans la quête de modèles fournissant un niveau d'abstraction adéquat. En conséquence, le besoin en outils génériques permettant leur déploiement à large échelle augmente de concert.

Le modèle de programmation chimique est un langage à base de règles dont le modèle d'exécution est inspiré par les processus chimiques. Dans ce rapport, en nous basant sur le modèle chimique, nous explorons l'exécution distribuée des langages à base de règles en déployant un environnement d'exécution chimique à large échelle.

L'architecture proposée combine une couche de communications pair-à-pair avec un protocole adaptatif de capture des données et un mécanisme efficace pour la détection de la terminaison. Nous décrivons le prototype logiciel développé et qui implémente l'ensemble de ces concepts. En nous appuyant sur un ensemble d'expériences menées à large échelle, nous confirmons les analyses de complexités précédemment menées, et montrons la viabilité d'un tel modèle de programmation à large échelle.

Mots-clés : Modèles de programmation à base de règles, calcul chimique, déploiement, expériences à large échelle

Contents

1 Introduction **3**
1.1 Chemistry-inspired Rule-based Programming 4
1.2 Motivation Example 5
1.3 Contribution and Organisation of the Paper 6

2 Platform Overview **7**
2.1 Initialisation 7
2.1.1 Data Distribution. 7
2.1.2 Meta-molecules. 8
2.2 Execution 9
2.2.1 Random Meta-molecule Fetch. 9
2.2.2 Search for Candidates. 10
2.2.3 Molecule Capture and Reaction Execution. 10
2.2.4 Execution of Multiple Rules. 11
2.3 Termination 11
2.4 Complexity Analysis 11
2.4.1 Execution Time Analysis. 11
2.4.2 Network Traffic Analysis. 12

3 Software prototype **12**
3.1 Entities 12
3.2 Execution Cycle 13
3.2.1 Initialisation. 13
3.2.2 Execution. 13
3.2.3 Termination. 14
3.3 Optimisations 14

4 Evaluation **15**
4.1 Test Programs 15
4.1.1 Highly-parallel Programs. 15
4.1.2 Producer/Consumer Programs. 15
4.2 Experimental Results 16

5 Related Works **17**

6 Conclusion **18**

1 Introduction

One challenge of distributed systems stands in finding *good* abstractions to *program* them. The emergence of a distributed computing platform calls for adequate programming models able to simply but fully leverage its capacities. The global computing platform which is today emerging on top of the Internet allows to interconnect a virtually infinite number of computing devices, which, aggregated, represent a tremendous computing power. However, due to the scale, dynamics and heterogeneity of such a platform, actually leveraging this power remains a wide open issue.

Abstracting out the technical details of the low-level machinery of the platform appears to be a prerequisite to actually being able to efficiently compute over it. In other words, the logic of the computation (which does not change, whatever the underlying platform characteristics are) should be separated from its (technical, low-level) implementation.

This situation advocates the use of declarative programming [15], whose goal is to separate the logic of a computation (*"what we want to do"*) from its control (*"how to achieve it"*), thus making it possible to simply specify the computation we want to run in a distributed platform. More precisely, while the "what" is to be defined by the programmer, the "how" becomes implicit, hidden in the system. In particular, rule-based programming, where the logic is expressed as a set of *rules*, is very attractive for parallel and distributed systems, as the parallelism and distribution, and their intrinsic difficulties are hidden to the programmer.

Recently, a lot of work has gone into showing how to concretely apply rule-based programming to the specification of distributed systems. For instance, in [11], it has been shown how communication protocols and peer-to-peer applications can be specified using a rule-based language. In [2], the same programming style is applied to web-based data management. On the computing side, rule-based programming was also used as a building block for workflow management systems, for instance in works such as [22, 13].

1.1 Chemistry-inspired Rule-based Programming

In this paper, we focus on the chemical programming model, a powerful rule-based programming model, which takes its inspiration in the chemical metaphor. It associates rule-based programming with an implicitly-parallel runtime. The chemical programming model has been advocated as one of the most promising paradigms to be studied in such a quest [5, 9, 17, 18].

Metaphorically speaking, in such a model, a program is envisioned as a *chemical solution* where molecules of data float and react according to some *reaction rules* specifying the program, to produce new data (the products of reactions). More formally speaking, it relies on *concurrent multiset rewriting*: the solution is a multiset of objects/molecules, and reactions are rewriting rules to be applied on it. At run time, these reactions can be triggered concurrently and reactions are carried out until the state of *inertia* has been reached — a stable state in which no more reactions are possible. Note that, following the philosophy of declarative programming, the order in which rules are to be triggered is not specified. In other words, it is left to the implementer. In this area, the Higher-Order Chemical Language (HOCL) is a full-featured rule-based language [4]. It provides the higher order: rules themselves are molecules in the multiset and can be consumed or produced dynamically at runtime. Hence, one can model programs able to evolve at run time. In HOCL, reaction rules are of the form:

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

where P is the pattern of reactants, V is a condition on them and M is the product of the reaction. Note that the applied rule itself is not deleted in the reaction. An HOCL program is a solution of molecules, that is to say, a multiset of non-ordered atoms (A_1, …, A_n) which can be constants (integers, booleans, *etc.*), sub-solutions (denoted $< M_i >$), tuples (denoted $M_1 : M_2 : \cdots : M_n$) or reaction rules. Let us first illustrate the paradigm with a simple HOCL example which counts, in parallel, the number of characters in a multiset of words:

$$\begin{array}{l}\textbf{let}\ \text{count} = \textbf{replace}\ s :: \mathit{string}\ \textbf{by}\ \mathit{len}(s)\ \textbf{in}\\ \textbf{let}\ \text{aggregate} = \textbf{replace}\ x :: \mathit{int}, y :: \mathit{int}\ \textbf{by}\ x + y\ \textbf{in}\\ < \text{“}\mathit{nel}\text{”}, \text{“}\mathit{mezzo}\text{”}, \text{“}\mathit{del}\text{”}, \text{“}\mathit{cammin}\text{”}, \text{“}\mathit{di}\text{”}, \text{“}\mathit{nostra}\text{”}, \text{“}\mathit{vita}\text{”}, \mathit{count}, \mathit{aggregate} >\end{array}$$

The rule *count* replaces a string by its length. The *aggregate* rule produces the sum of two consumed integers. At run time, these rules are triggered repeatedly and concurrently, the first one producing inputs for the second one. Note that the order in which rules are triggered is not deterministic; only the atomic capture of reactants is ensured. A possible succession of states is the following, the last one being inert and "$\downarrow *$" denoting the fact that several rules are applied, possibly in parallel:

$$< \text{“}nel\text{”}, \text{“}mezzo\text{”}, \text{“}del\text{”}, \text{“}cammin\text{”}, \text{“}di\text{”}, \text{“}nostra\text{”}, \text{“}vita\text{”}, count, aggregate >$$
$$\downarrow *$$
$$< 3, 3, 4, 6, \text{“}mezzo\text{”}, \text{“}di\text{”}, \text{“}nostra\text{”}, count, aggregate >$$
$$\downarrow *$$
$$< 10, 6, 5, 2, 6, \text{count}, \text{aggregate} >$$
$$\downarrow *$$
$$< 29, \text{count}, \text{aggregate} >$$

1.2 Motivation Example

Let us now illustrate the model in context by providing an example of an *autonomic* server, *i.e.,* a server able to run in the most efficient and reliable way given the characteristics of the targeted platform exposed above. More concretely speaking, considering a simple task continuously requested by some clients, and for which several implementations (or *services*) exist on the platform, we want to: (1) select the best service implementation according to some predefined policy, (2) change the optimisation policy dynamically if the criterion changes, and (3) recover automatically after the failure of the service implementation currently in use.

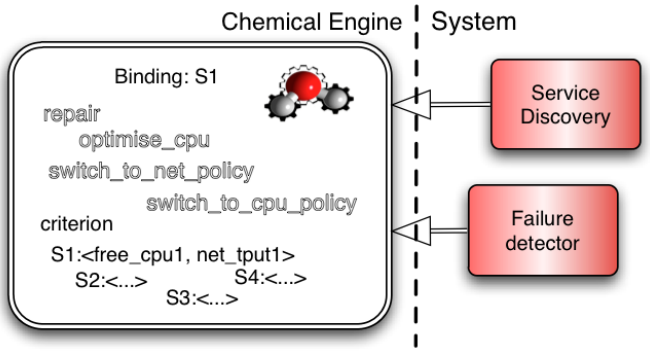


Figure 1: An HOCL-based autonomic service.

Figure 1 illustrates an HOCL-based implementation of this self-adaptive service. The multiset (on the left-hand side of the figure) interfaces with two system components that achieve two respective tasks. The first one discovers available service implementations and injects them in the solution as S_i:<*free_ cpu*$_i$, *net_ tput*$_i$> molecules. Each such injected molecule represents a concrete service able to perform the task asked by clients. Apart from its identifier (S_i), such a molecule also contains a sub-solution indicating some dynamic performance indicators. The list of indicators can be arbitrarily extended. The second connected system service detects the failure of the currently active service (denoted by the *Binding:*S_i molecule) and introduces a *"failure_ detected"* molecule upon detection. The rules that drive the execution follow. First, the *repair* rule needs the presence of the specific molecule indicating a failure. Once triggered, it binds the task to another service implementation:

let *repair* = **replace** *Binding:*S_i, S_i:<ω_i>, S_j:<ω_j>, *"failure_ detected"*
by *Binding:*S_j, S_j:< ω_j >

Let us now review an optimising rule, named *optimise_ cpu*:

let *optimise_cpu* = **replace** *Binding:*S_i, S_i:<*free_cpu*$_i$, *net_tput*$_i$>, S_j:<*free_cpu*$_j$, *net_tput*$_j$>
by *Binding:*S_j, S_i:<*free_cpu*$_i$, *net_tput*$_i$>, S_j:<*free_cpu*$_j$, *net_tput*$_j$>
if (*free_cpu*$_j$ > *free_cpu*$_i$)

A reaction following the *optimise_cpu* rule is triggered when a service molecule S_j with a better CPU availability is found in the multiset. This rule corresponds to the decision taken by the system to select services based on their CPU availability. Similarly, an *optimise_net* rule may consider the services' network capabilities.

Having different policies brings more flexibility to the adaptation but creates the need for dynamic switching from one to the other based on a criterion, in this case meaning *optimise_cpu* might have to be put aside in favour of *optimise_net*. This can be achieved through the higher order, using the following rule, which replaces an optimising policy by another one as soon as it detects the criterion has changed:

let *switch_to_net_policy* = **replace** *optimise_cpu*, *criterion::string*
by *optimise_net*
if (*criterion* = *"Net"*)

Note that, as illustrated on Figure 1, other rules, for instance *switch_to_cpu_policy*, can be similarly constructed and introduced in the solution concurrently. They can coexist smoothly in the solution, as the criterion can take only one value at a time, preventing concurrent reactions of contradictory switching rules. Finally, note that for the sake of simplicity, the example deals with only one service. However, it can be easily extended so as to deal with many services distributed over the nodes of a large scale platforms, each area of the platform having its own criteria and policies changing concurrently.

1.3 Contribution and Organisation of the Paper

Our goal is to provide a generic distributed platform dedicated to the execution of chemical programs. We envision a high number of nodes willing to collaborate, with each collaborating node equipped with an engine executing rules on the molecules they hold. The four following issues need to be tackled:

1. *Abstract communications.* Each node has to be able to communicate with every other node.
2. *Discover molecules.* Molecules are now dispatched over the network, meaning suitable reaction candidates have to be found efficiently in spite of the scale of the platform.
3. *Capture molecules atomically.* Once the appropriate molecules have been located, a node must grab *all of them* atomically, as other nodes may try to fetch them as well at the same time.
4. *Detect the program termination.* To secure the termination of a program, we need to ensure to detect that no more reactions are possible. This detection, when done in a centralised way, has a combinatorial complexity, as every combination of molecules has to be checked against the rules (yielding $m!$ tests for a program containing m molecules). This suggests that relying on intelligent information retrieval techniques is mandatory in order to circumvent the problem.

Contribution. This paper builds upon the preliminary work in [19], which gave a first conceptual view of our proposal and which is summarised in Section 2. In this proposal, a distributed hash table (DHT)-based framework solving the four previously mentioned issues was proposed. The

resolution of the first two items highly relies on the presence of the DHT, while the resolution of the third one relies on a recently proposed protocol to capture several objects atomically in concurrent settings. Finally, the inertia detection is based on a second information retrieval layer built on top of the DHT. In [19], the validation of such a platform was only based on analysis. The present paper, in contrast, after reviewing the concepts proposed, focuses on the experimental validation of the architecture. Firstly, a software prototype implementing the concepts is detailed. Secondly, real deployments of chemical programs undertaken are presented and their results are discussed. In other words, this paper provides the *"how"* of distributed concurrent rule-based programming, allowing programmers to concentrate on the *"what"*.

Organisation of the Paper. Section 2 summarises the global architecture, its data structures and algorithms, and its analytical validation. The software prototype is presented in Section 3, and the conducted experiments are discussed in Section 4. Section 5 presents the other few previous attempts at deploying the chemical model on real platforms. Section 6 concludes.

2 Platform Overview

The overview of the distributed execution platform is depicted on Figure 2[1]. The platform itself is built on top of an overlay network, organising the participants in a ring-like structure. While we have chosen to rely on Pastry [20], DHTs with different topologies [16] could be used. The DHT concept secures independence from the underlying environment and partially solves the scalability issue — nodes are able to communicate efficiently regardless of their number, typically in a logarithmic number of hops in the overlay. Let us assume the program to be executed is initially held by an *external application*. It contacts any node it knows of in the DHT and transfers it the program. Alternatively, the external application may itself join the platform and partake in its program's execution. The platform is intended to execute many programs concurrently. For the sake of parallelism and load balancing, each of them can get a different *entry point* in the system.

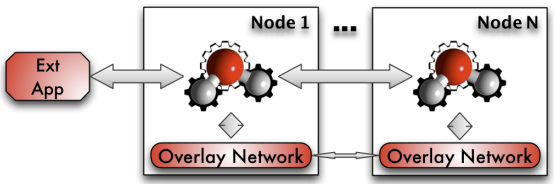


Figure 2: The platform.

The remainder of this section details the initialisation, execution and termination of a program running on this platform.

2.1 Initialisation

The node contacted by the external application is called the *source node* since it represents the data's entry point in the system. Additionally, upon termination, the inert solution, *i.e.* the result, is transferred from the source node to the external application.

2.1.1 Data Distribution.

After receiving the data from the application, the source node scatters the data molecules across the system according to their hash values. The cryptographic hash function of the underlying

[1]This section is a summary of the platform described in [19]. However, for consitency and self-containment reasons, it is reviewed in a comprehensive manner.

DHT guarantees uniform dispersion with high probability (w.h.p.), in this way globally load-balancing access to molecules. Molecules are routed concurrently according to Pastry's routing scheme, in $O(\log n)$ hops, where n denotes the number of nodes. In this way, each node holds a subset of the program's data molecules with high probability if the number of molecules is high enough, and all of the rules, enabling a high level of parallelism and concurrency in performing reactions. By tracing the molecules' paths, a tree, rooted at the source node, is created. Once all of the molecules have been routed, the source node uses this *multicast tree* to diffuse the rules contained in the program. Furthermore, this multicast tree will be used during the termination phase to collect the resulting inert solution.

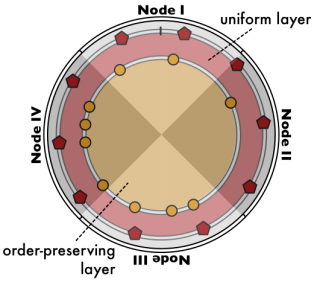


Figure 3: Double layer: the key space of the uniform layer coincides with that of the order-preserving layer. The alternating grey and white regions designate responsibility areas of different nodes, in a four node network.

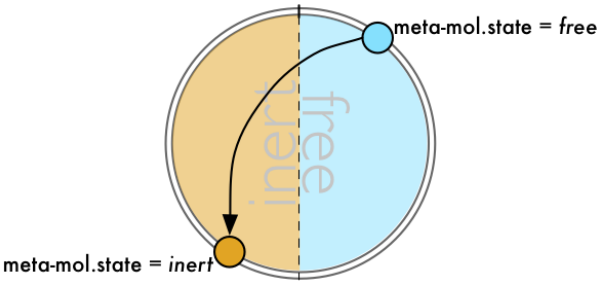


Figure 4: Order-preserving layer: as a meta-molecule's state changes, it repositions itself in the second layer.

2.1.2 Meta-molecules.

Since molecules are spread throughout the system, nodes must be able to find suitable candidates for reactions. In order to efficiently detect inertia, and consequently increase the platform's scalability, with regard to both the number of nodes as well as the number of molecules, on top of the existing DHT layer (referred to as the *uniform layer*), we use a second DHT layer (referred to as the *order-preserving layer*). It contains *meta-molecules*, which are placed around the key space in an order-preserving manner, allowing participants to use range-query techniques [8, 21] to search for the existence of a particular molecule, or a set thereof, during the execution phase. The organisation and physical placement of the two layers are illustrated in Figure 3.

A meta-molecule is a simple meta-data object affirming a molecule's existence in the system, and providing information about its original identifier and its current state. Meta-molecules offer a lightweight indexing and look-up mechanism because instead of exchanging heavier objects like molecules, nodes are able to query and exchange only their lighter counterparts — the meta-molecules — until they find the reactants they have been looking for.

Each node produces meta-molecules based on its local molecules and routes them according to their order-preserving identifier. Each molecule is associated a state in its meta-molecule. Initially, a meta-molecule's state is set to *free*, indicating that nodes can freely take the molecule it describes and combine it with other molecules to perform reactions. At a later stage, during execution, a meta-molecule's state may be set to *inert*, which denotes that a suitable combination for its molecule has not been found thus far, as discussed in more detail later.

The order-preserving layer is split in two parts: the one containing only *free* meta-molecules, within the id range $[0, \frac{ks}{2} - 1]$, and the other consisting of only *inert* meta-molecules, within the

id range $[\frac{ks}{2}, ks-1]$, where ks is the size of the key space. Both halves of the key space are organised in the same way: the position of a meta-molecule is based on the total ordering of values of a specific molecule type. A meta-molecule's hash identifier is calculated as:

$$id = \begin{cases} \frac{cv}{\|\mathcal{V}\|} * \frac{ks}{2} - 1 & \text{for } state = free \\ \frac{ks}{2} * (1 + \frac{cv}{\|\mathcal{V}\|}) - 1 & \text{for } state = inert \end{cases}$$

where cv is the molecule's value's cardinal position in the total order and $\mathcal{V}$ is the set of all possible values the molecule can have. As a consequence, when a meta-molecule's state changes, its identifier is recalculated, relocating it to the *other* half of the key space, as illustrated by Figure 4.

2.2 Execution

The distributed platform presented adopts intelligent reactant searching, in which the system is explored for molecules with specific properties matching a rule's pattern and condition, such as *an integer greater than 3*, allowing it to efficiently detect inertia.

Algorithm 1: Main execution loop.

```
1  while not inert do
2      meta_mol1 = random_mol(state = free);
3      if meta_mol1 = null then
4          break;
5      meta_mol2 = find_candidate(meta_mol1,
       rule);
6      if meta_mol2 = null then
7          meta_mol1.state = inert;
8          store(meta_mol1);
9          continue;
10     if grab_molecules(meta_mol1,meta_mol2)
       then
11         execute_reaction(rule, mol1, mol2);
12         store(new_mol1, new_mol2);
13         store_ack(meta_new_mol1,
           meta_new_mol2);
14         remove(meta_mol1, meta_mol2);
```

The main execution loop, executed by every node, is described in Algorithm 1. For the sake of clarity, the algorithm is presented in a simplified form, in which only one rule involving a pair of molecules is considered. Nevertheless, it is easily expandable to multiple rules and multiple molecules per rule. It consists of three steps: (i) getting a random meta-molecule and testing inertia (lines 2—4), (ii) finding a candidate it can react with (lines 5—9) and (iii) atomically grabbing the corresponding molecules and performing the reaction (lines 10—14).

2.2.1 Random Meta-molecule Fetch.

During the first step, a node tries to obtain a random meta-molecule, the state of which is set to *free*. `random_mol` guarantees a *free* meta-molecule will be returned, in case one exists. If, on the other hand, no meta-molecule can be found, it means the system could not find any candidate for the currently present molecules, implying their states are set to *inert*. This signals to the requesting node that inertia has been reached. It then stops executing the main loop (line 4). The actions taken by the platform once inertia has been detected on a single node are laid out in Section 2.3.

`random_mol` generates a random identifier within the range $[0, \frac{ks}{2} - 1]$ since the node is looking for a *free* meta-molecule. Random fetches improve parallelism and load balancing since, in doing so, nodes try to fetch different meta-molecules. The requesting node sends a *METAMOL_REQ* message request comprising the generated identifier, the node's identifier and the range to be scanned — the whole half key space —, to the corresponding node. The receiver of the request checks its local meta-molecules and returns it the one with the closest identifier. The method is exemplified in Figure 5.

In case the receiver of the request does not hold any *free* meta-molecules, it splits the search range into two parts: $[range_beginning, min_id - 1]$ and $[max_id + 1, range_end]$, where

$[min_id, max_id]$ denotes the receiver's responsibility area. It then generates one random identifier for each range and sends two *METAMOL_REQ* message requests in parallel. This process continues until a meta-molecule has been found or until the whole half key space has been searched with no result. In the latter case, a *NO_METAMOLS* message is sent to the original requester, after which the termination phase is triggered on both the original requester and the original receiver of the request.

2.2.2 Search for Candidates.

Obtaining a free meta-molecule triggers the second execution step (lines 5—9). The node now asks the system to find it a suitable meta-molecule by supplying the meta-molecule found in step one and the rule which needs to be applied on the molecules to the `find_candidate` routine (line 5). This routine systematically searches for a meta-molecule matching the provided rule's pattern and reaction condition.

The rule, of the form **replace** P **by** M **if** V, is analysed and the type of the candidate is first extracted from the pattern P. Next, the random meta-molecule is introduced in the reaction condition V; the cardinal position of the value replaces its symbol. Such a modified rule is then tested to establish the values which would lead to a positive evaluation of the reaction condition. These values are aggregated into ranges of values and enclosed each in a *METAMOL_REQ* message along with the desired type.

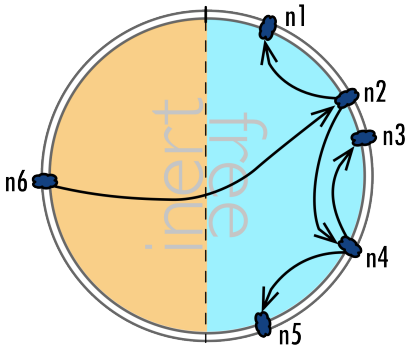


Figure 5: Meta-molecule fetch example: $n6$ asks $n2$ for a random meta-molecule, which then forwards the request to $n1$ and $n4$. The process is repeated until a *free* meta-molecule is found.

The process of finding a matching meta-molecule in the system conforms to the previously described method of locating random meta-molecules, only this time the ranges cover the whole key space. Thus, at least two *METAMOL_REQ* messages are sent: one targeting *free* meta-molecules and the other *inert* ones. If neither of the requests returns a meta-molecule, the requester sets the initial random meta-molecule's state to *inert* and stores it in the second half of the key space.

Note that, in case a range is fragmented, each part is enclosed in a different *METAMOL_REQ* message. Thus, the requester may receive more than one meta-molecule suitable for a reaction. The node then chooses randomly one of the arrived meta-molecules, discarding the rest.

2.2.3 Molecule Capture and Reaction Execution.

Step three (lines 10—14) concludes an execution loop iteration. The node tries to *capture* the molecules described by the previously obtained meta-molecules. Given the fact that a molecule can be consumed only once, *i.e.* it can be used in at most one reaction during its lifetime, it is imperative that nodes grab all of the molecules needed in an atomic fashion. For this task, we rely on a capture protocol able to dynamically self-adapt in regard to the probability of conflicts when trying to capture the molecules. As the capture is not our primary concern here, we refer the reader to [6] for more information about the protocol.

If the `grab_molecules` routine succeeds, it ensures no other node can obtain these molecules, triggering the actual execution of the reaction, after which the meta-molecules describing the newly created molecules are produced. The new molecules are then sent to their respective nodes based on their hash identifiers. To store the meta-molecules, the `store_ack` procedure is

used, which blocks the execution until the node receives the confirmation of their arrival at their respective destinations. Only then the old meta-molecules are removed from the second DHT layer, as the molecules they represent do not longer exist.

2.2.4 Execution of Multiple Rules.

Algorithm 1 outlines the steps only for one rule and two molecules. However, in its full form, a node chooses randomly one of the rules received from the source node before the beginning of step one. Additionally, if the selected rule takes more than two arguments, the instructions of the second step (lines 5—9) are iteratively repeated until the right amount of candidate meta-molecules has been obtained. In case suitable candidates cannot be found, step two is repeated for all the rules a node possesses before marking the meta-molecule found in step one as *inert*. Naturally, when a one-argument rule is executed, the candidate search step is skipped; the node proceeds directly to the molecule capture step after obtaining a random *free* meta-molecule.

2.3 Termination

Inertia has been detected once `random_mol` (Algorithm 1, line 2) can no longer find a *free* meta-molecule in the system[2]. This marks the end of execution and the beginning of the termination phase.

The node which perceives inertia has been reached propagates up and down the tree *TERMINATION* messages. Upon the receipt of this message, a node stops the execution phase and awaits the molecules held by its children and then sends them to its parent, together with the molecules it holds. Eventually, all of the molecules will reach the source node, which then transfers the now inert solution to the external application.

It is a seemingly sequential process which spreads from a node up and down the tree. However, since all of the nodes equally participate in the execution process, there are going to be multiple nodes which will detect inertia at more or less the same time, effectively speeding up the termination phase.

2.4 Complexity Analysis

We now provide a brief complexity analysis of the time and network costs of running a chemical program on top of the proposed platform. Due to the versatility of the chemical paradigm and the volatility of the execution (asynchronism, conflicts during molecule captures), a precise analysis of the general case is not feasible. In the following, we present a static analysis of a chemical program containing only reaction rules reducing the number of molecules. This behaviour mimics most data-processing applications and services, where multiple input values are processed to produce one output value. We assume the presence of a single rule with r arguments acting upon m molecules in a system comprised of n nodes which are uniformly arranged across the key space, with $m \gg n \gg r$. The rule produces a single molecule as its output.

2.4.1 Execution Time Analysis.

The cost of the initial dissemination depends on the number of molecules to be disseminated, thus running for $O(m)$ time. The termination phase is conditioned by the number of nodes having to return their molecules, putting its cost to $O(n)$.

[2]The fact that being unable to find a *free* meta-molecule ensures that the inertia has been globally reached was formally established in [19].

The number of loops of Algorithm 1 each node will do is proportionate to $\frac{m}{r}$. However, because $m \gg n$, at the beginning of the execution every node will be able to capture an exclusive set of r molecules (w.h.p.). As the computation progresses, more and more conflicts will arise, linearly decreasing the number of nodes able to complete a reaction, until there are only r molecules left for which all of the nodes will compete. Consequently, on average $\frac{n}{2}$ nodes perform a reaction in each iteration, while the rest aborts theirs. Thus, each node will loop through Algorithm 1 $\omega = \frac{2m}{rn}$ times, consuming $O(\frac{m}{n})$ units of time.

2.4.2 Network Traffic Analysis.

The cost of the initial dissemination in terms of messages is $O(m \log n)$, while that of the termination phase is $O(n)$.

In the worst-case scenario, which happens towards the end of the execution phase when there are only a few *free* meta-molecules left, `random_mol` and `find_candidate` generate at most $2 \log n + n + \log n$ messages each. $2 \log n$ messages are needed for sending the request and receiving the response; n messages are spent to search the key space and $\log n$ messages are used for returning a molecule to the first node in the key space contacted by a requester.

In this worst-case scenario, the capture protocol needs $6r \log n$ messages to complete, as devised in [6]. As explained above, $\frac{n}{2}$ nodes perform reactions, completing all the capture protocol. This situation produces $10r\frac{n}{2} \log n$ messages in total. The nodes which have successfully grabbed the r molecules generate further $(3+r)\frac{n}{2} \log n$ messages for storing the new molecules and meta-molecules and removing the consumed ones. Since the number of such cycles needed to reach inertia equals to ω, the cost of the execution phase is $O(mn \log n)$.

The execution phase generates the highest number of messages when compared to the other phases. Consequently, we approximate the cost of a complete program execution in terms of network cost to be $O(mn \log n)$. Bearing in mind the factorial complexity of the traditional centralised method, this analysis shows the plausibility of executing chemical programs on the proposed platform with a network traffic overhead proportional to the program's size and to the number of nodes, establishing the benefits of such a distributed platform.

3 Software prototype

Following the model of the platform laid out in Section 2, we developed a fully-functional software prototype in Java[3]. Figure 6 shows its logical view.

3.1 Entities

It is composed of five entities:

- **Overlay Network.** The abstraction from the underlying physical network is handled by this entity. Its main component is FreePastry [1], an open-source DHT coded in Java and developed and maintained by the authors of Pastry.
- **Molecule Holder.** This entity is the implementation of the uniform DHT layer and as such it serves as a container for molecules held by the node. In order to store, index and retrieve molecules more easily, they are grouped by their molecule types and sorted based on their hash identifiers. The molecule holder is contacted during the atomic capture step

[3]The sources are available in the *branches/devel-distrib* directory of the *svn* repository located at *http://gforge.inria.fr/scm/?group_id=2125*.

and is in charge of deciding whether and to which node a molecule it holds will be given, according to the capture protocol.

- **Meta-molecule Holder.** Analogously, this entity represents the implementation of the order-preserving DHT layer and is, thus, a repository of meta-molecules. It manages the insertion, retrieval and deletion of meta-molecules requested by other nodes. Note that when a retrieval request is received, the meta-molecule is not removed. Instead, its copy is returned to the requesting node. Moreover, it handles random meta-molecule fetches and candidate requests. If it cannot satisfy the request, it communicates with meta-molecule holders on other nodes to complete it, as specified by the algorithm devised previously.

- **Tree Manager.** The multicast tree created during the initialisation phase is constructed by this entity. It maintains the node's *local state* (consisting of its parent and children) and uses it to spread the rules down the tree and to send its and its children's remaining molecules to its parent.

- **Central Unit.** This is the main entity in the prototype. It communicates with the application (taking the program to execute from it and returning the inert result to it) and executes the main execution loop (Algorithm 1).

Each of the entities cooperates with the entities directly above and below it portrayed in Figure 6.

3.2 Execution Cycle

3.2.1 Initialisation.

A first step is for the application to transfer the program to execute to the central unit. It then hashes the molecules and dispatches them to the overlay network, which spreads them in the uniform layer. During this period, the tree manager monitors the overlay network traffic and when it stumbles upon a molecule, it adds the destination node to its local state. Once the molecules have been disseminated, the central unit hands the rules over to the tree manager, which sends them to its children in the local state.

On the receiving end, when a node receives a molecule, it stores it in the molecule holder. This entity creates a meta-molecule for each held molecule and routes it in the order-preserving layer through the overlay network. The node then receives the rules to execute, upon which the tree manager completes its local state by assigning the node's parent in the tree (the node which has sent it the rules). Now the nodes are ready to start the execution.

3.2.2 Execution.

At this point, the central unit on each node starts the main execution loop. It asks the meta-molecule holder to find it a random meta-molecule in the network. A rule is randomly chosen from which the type of one of its reactants is extracted. The meta-molecule holder then sends out requests in the *free* half of the order-preserving layer to find such a meta-molecule. This process is repeated for each rule until a meta-molecule has been returned. Then, the central unit *translates* the pair

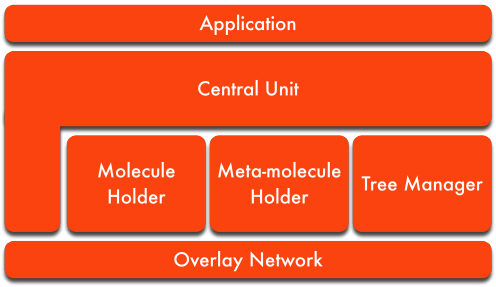


Figure 6: Logical view of the entities forming the prototype.

(`rule`, `meta − molecule`) into a range query request by injecting the meta-molecule's identifier in the rule. The request is, again, handed over to the meta-molecule holder which tries to find a candidate meta-molecule satisfying the range query in the order-preserving layer. The process of searching for a candidate is repeated for as many reactants the rule needs, each time introducing the newly acquired meta-molecule into the rule and constructing a new range query for the next candidate to be located. If the candidates cannot be found, a delete request is sent to the random meta-molecule's holder. Its state is then changed to *inert* (changing its identifier) and stored in the second DHT layer.

Following Algorithm 1, the final step of the execution phase consists in grabbing atomically the molecules and performing the reaction. For capturing the molecules, the capture protocol in [6] was implemented. In this protocol, the central unit plays the role of the molecule requester. It extracts the identifiers of the molecules to grab and sends the fetch requests to the corresponding nodes. Their molecule holders then evaluate each its own request and decide whether to send back the molecules. Only in case all of the molecules have been received the node performs the reaction. Then, as per Algorithm 1, the products of the reaction and their meta-molecules are stored in the DHT (each in their respective layer) and delete requests for the consumed molecules' meta-molecules are sent.

3.2.3 Termination.

Once there are no more *free* meta-molecules in the order-preserving layer, the nodes enter the final, termination step of the execution. A node starts this phase when its meta-molecule holder is not able to find a random meta-molecule. At that point, the tree manager awaits the node's children's molecules. These are then combined with the molecules held by the molecule holder and sent to the parent up the tree. Finally, the central unit of the source node delivers the inert solution to the requesting application.

3.3 Optimisations

Even though the prototype follows the description of the system model discussed in earlier sections, it carries two slight improvements dealing with local meta-molecule search and meta-molecule retrieval. Both of the enhancements are implemented in the meta-molecule holder.

- The first optimisation exploits the principle of *locality*: whenever the central unit requests a meta-molecule, the meta-molecule holder first checks whether it can satisfy the request right away without querying other nodes. This method is applied to both random meta-molecule and candidate search requests. At the beginning of the execution, nodes which are located in the *free* half of the key space will be able to benefit directly from it, seeing that during that time most of the meta-molecules' states are set to *free*, enabling nodes to pick a random meta-molecule from their local meta-molecule holders. Towards the end of the execution, on the other hand, nodes located in the other half of the key space can benefit from the principle during the candidate search step, since most meta-molecules will be labelled as *inert* at that point.

- The second improvement is introducing a small decision-making mechanism into the meta-molecule holder. Whenever it receives a retrieval request, it tries to return the meta-molecule closest to the requested identifier. It is thus possible for the same meta-molecule to be sent to more than one node. Even though the capture protocol assures a molecule is going to be consumed in only one reaction, giving the same meta-molecule to different nodes

generates superfluous network traffic as some grab requests will be aborted. Therefore, the meta-molecule holder keeps track of the number of times each meta-molecule has been handed out to nodes. Doing so, it is able to return the meta-molecule which satisfies the request criteria but has been handed out less times than other meta-molecules. Such a slight refinement ultimately minimises the number of conflicts between nodes over molecules and consequently the network overhead due to capture aborts.

4 Evaluation

In this section we present the evaluation of the software prototype described above. To better capture the viability of the proposed platform we tested it on two programs with different properties. They are presented in Section 4.1, while the results obtained are detailed in Section 4.2.

4.1 Test Programs

We present now two distinct classes of programs on which we tested the proposed platform: highly-parallel and producer/consumer ones.

4.1.1 Highly-parallel Programs.

In such applications, the same operation is applied to the whole of the input data. It is thus interesting to investigate the behaviour of a decentralised execution environment when faced with such programs. We chose to represent them with `GetMax`. It is a simple single-rule program containing only integer molecules. The rule consumes two such molecules and produces a new one holding the higher value of the two. In addition to being highly parallel, this program resembles most data-processing applications, where multiple input variables are processed to give an output.

From the point of view of the execution of chemical programs, the interest of `GetMax` stands in that it represents a program with a decreasing complexity — the number of molecules declines with every reaction. Furthermore, regardless of the course of the execution the total number of reactions performed during its execution is always constant.

4.1.2 Producer/Consumer Programs.

The second category of applications is the producer/consumer one. The interest in this class stands in that these programs impose the sequentiality of events — a producer has to produce the input for the consumer — on an implicitly parallel paradigm executing in a decentralised environment. While highly-parallel programs designate data-processing applications, producer/consumer ones can be interpreted as their temporal compositions — *workflows*.

The experiments were conducted with `StringManip` — a program comprised of two rules manipulating string molecules. The logic of `StringManip` consists in splitting and packing together string molecules in such a way that the resulting string molecules have a predefined length, denoted λ. The first rule, *SplitStr*, consumes one molecule whose string's length is greater than λ and produces two molecules; one is composed of the first λ characters of the original molecule's string, while the other contains its remainder. The second rule, *ConcatStr*, takes two molecules as input and outputs one which is their concatenation. Thus, *SplitStr* produces the molecules which are going to be consumed by *ConcatStr*.

The course of the execution of `StringManip`, as well its outcome, is non-deterministic. While it is known that at the end of the execution the molecules' strings are going to have a length of λ,

their contents depend on the succession of reactions performed by the system, which is influenced by the asynchronous nature of the platform. In other words, the outcome is conditioned by the reactions performed by each node, their input molecules, and the order in which they actually take place. Hence, the number of reactions done throughout the execution varies from run to run. Furthermore, the two rules are *circularly dependent* on each other — *ConcatStr* might produce molecules which are going to be consumed by *SplitStr* —, in this way bringing a partial *sequentiality* into the program.

4.2 Experimental Results

We conducted experiments using Grid'5000[4] [7], the French national grid test-bed. The nodes were spread across nine geographically-distant sites. Each experiment was run ten times, and here we present the average of the values obtained. For every run the nodes were randomly chosen to obtain different network topologies.

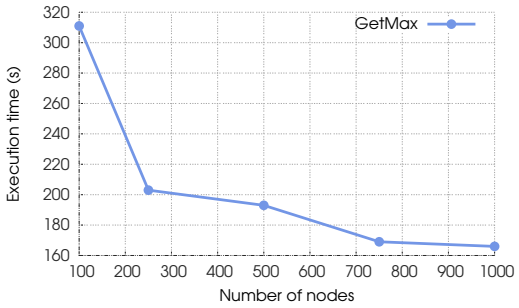


Figure 7: Execution time of `GetMax` containing 50,000 molecules.

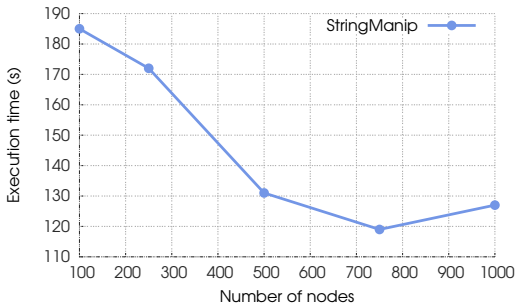


Figure 8: Execution time of `StringManip` with 20,000 molecules.

Experiment 1. Firstly, we evaluated the viability of the platform by executing the two programs while varying the number of nodes participating in the execution. Figures 7 and 8 show the execution times obtained for `GetMax` and `StringManip`, respectively.

In both cases there is a decrease in execution time when increasing the number of nodes carrying out the computation, which is in compliance with the results of the complexity analysis from Section 2.4.1. Moreover, significant speed-ups were obtained. However, one can notice that the speed-up obtained for `GetMax` is greater than that for `StringManip`. This is due to the difference of the programs' characteristics. On the one hand, the number of molecules in the system strictly decreases after each reaction when executing the `GetMax` program, while the trend is not known for `StringManip` — it may stay constant, decrease or increase. On the other, the execution time of `StringManip` depends on the sequentiality of events: certain reactions cannot be carried out before others are. In contrast, `GetMax` is a highly-parallel program where the maximum possible number of reactions can be performed in any given point in time. Finally, the execution takes more time to complete for 1000 nodes than for 750 when executing `StringManip`. This is the result of the program's sequentiality: more nodes are in conflict over a subset molecules since not all available molecules can be used straight away, in this way prolonging the execution.
Experiment 2. During the execution of the programs we also monitored the generated network traffic. Figures 9 and 10 depict the total number of messages sent. Note that the number of messages in the case of `GetMax` is higher than that of `StringManip` due to the fact that there

[4] `http://www.grid5000.fr`

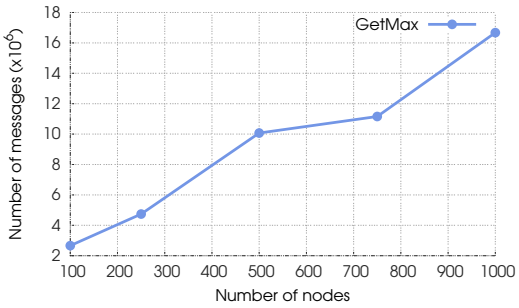


Figure 9: Number of messages sent during the execution of `GetMax`.

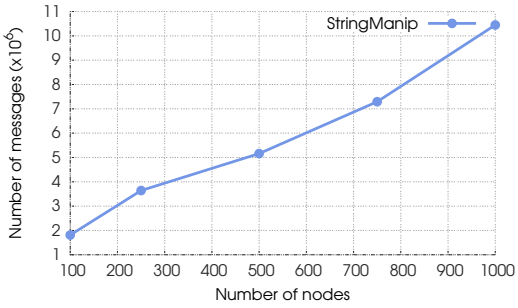


Figure 10: Number of messages sent during the execution of `StringManip`.

are more molecules in the initial solution of `GetMax`. Both of them show a linear augmentation in the number of messages, which conforms to the findings of the presented complexity analysis. We can see, however, that the curve for `GetMax` is steeper than that of `StringManip`. This is due to the fact that, because of the constant number of reactions, when there are more nodes involved in the computation there are more conflicts over molecules during the capture phase. In spite of this effect, one can notice that the actual number of messages per node declines with the growth of the network, which leads to the conclusion that the platform is scalable in terms of network load. We can thus conclude that the platform scales well.

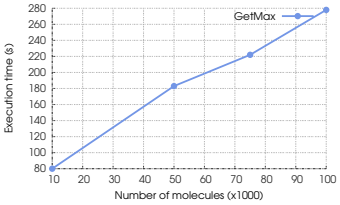


Figure 11: Execution time of `GetMax` on 500 nodes.

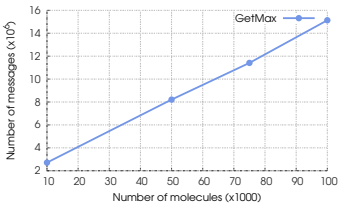


Figure 12: Number of messages sent during the execution of `GetMax`.

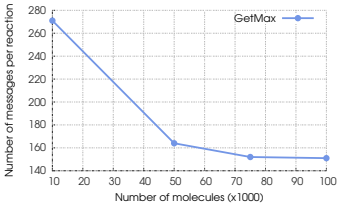


Figure 13: Number of messages sent per reaction during the execution of `GetMax`.

Experiment 3. In this experiment we fixed the number of nodes to 500 while varying the number of molecules contained in the `GetMax` program. Figure 11 shows that the execution time linearly grows with the increase of the size of the problem. The same effect can be observed when looking at the network traffic, depicted in Figure 12. Both figures confirm the analysis' findings: the system is scalable with regard to the size of the problem to be solved (*i.e.* the number of molecules). It is interesting to note that the number of messages needed to perform one reaction, illustrated in Figure 13, decreases. Indeed, when the number of molecules increases while keeping the number of nodes constant, there are less conflicts between nodes over molecules, and thus less communication is needed to carry out a reaction.

5 Related Works

In spite of the fact that the chemical model is implicitly parallel, and that it was proposed in an early form 25 years ago, not much work has been done on parallel or distributed execution of such

programs. The pioneering work of Banâtre *et al.* [3] provides two implementation methods of GAMMA programs based on a parallel machine. Each processor holds a molecule and compares it with the molecules of all the other processors. Two algorithms are proposed: (a) a synchronous one, where a centralised controller triggers each comparison step, and (b) an asynchronous one, in which molecules travel along a vector of processors, either until they react, or until they have returned to their starting point. This last algorithm was implemented on top of an iPSC hypercube with 16 processors. In the work by Linpeng *et al.* [12], a program is executed on MasPar MP1, a massively parallel machine, using the fold-over operation. The molecules are placed on a strip and folded over after each vertical comparison. At each step, the elements in the upper segment of the strip are compared in parallel to those in the lower segment. Recently, Lin *et al.* have showed that GAMMA can be used to model programs executable on a cluster exploiting GPU computing power [14]. Although these works present significant speed-ups, they consider the execution of particular chemical programs on very specific, small scale platforms. In this paper we focus on a generic runtime for large-scale heterogeneous environments.

Concerning the NP-completeness of the model's inertia detection mechanism, Gladitz and Kuchen [10] explored the possibility of reducing the number of steps needed to find suitable candidates and consequently detect inertia. They showed that by carefully examining the reaction condition and rearranging its terms based on the *relevance* of each variable, one is able to cut certain branches of the search tree, and in doing so considerably decrease the amount of computation required to find candidates and detect inertia.

Even though their shared-memory implementation shows significant performance improvements, the model cannot be exploited in practice since reaction condition examination is in itself an NP-complete problem.

The presented works regard GAMMA as a specification language: a program's behaviour is described using GAMMA and then written in an imperative language in order to be executed. This paper, however, proposes a distributed execution runtime which turns GAMMA's practical descendent, HOCL, from a specification language into an implementation language suitable for distributed execution.

6 Conclusion

Declarative programming has been recently identified as a promising, high-level model to develop distributed systems in a simple manner. However, this calls for mechanisms able to make it real over large scale platforms. In the area of declarative programming, concurrent rule-based programming (and its chemically-inspired representative) appears to be a highly expressive programming model offering an adequate level of abstractions in different areas of distributed computing.

Thus, the large-scale execution of chemical programs has to be tackled in order to provide the *"how"* to execute programs on such infrastructures and let programmers focus on the *"what"* to program, and in order to put the attractive characteristics of declarative programming into practice over large scale platforms.

This paper proposes a generic framework to solve this issue. On top of a two-layer distributed hash table, the framework relies on two distributed protocols, the first one capturing molecules in a highly concurrent system, the second one efficiently detecting *inertia*. These concepts have been used to implement a prototype, which was tested on a real-world test-bed. The experiments conducted corroborate the findings of the theoretical analysis about the sustainability of the proposed execution runtime.

References

[1] Freepastry. http://www.freepastry.org (June 2012)

[2] Abiteboul, S., Bienvenu, M., Galland, A., Antoine, E.: A rule-based language for web data management. In: Proceedings of the thirtieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems. pp. 293–304. PODS '11, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/1989284.1989320

[3] Banâtre, J.P., Coutant, A., Le Metayer, D.: A Parallel Machine for Multiset Transformation and its Programming Style. Future Generation Computer Systems 4, 133–144 (September 1988)

[4] Banâtre, J.P., Fradet, P., Radenac, Y.: Generalised Multisets for Chemical Programming. Mathematical Structures in Computer Science 16 (2006)

[5] Banâtre, J.P., Radenac, Y., Fradet, P.: Chemical specification of autonomic systems. In: 13th ISCA International Conference on Intelligent and Adaptive Systems and Software Engineering. pp. 72–79 (2004)

[6] Bertier, M., Obrovac, M., Tedeschi, C.: A Protocol for the Atomic Capture of Multiple Molecules on Large Scale Platforms. 13th International Conference on Distributed Computing and Networking (2012)

[7] Bolze, R., Cappello, F., Caron, E., Daydé, M., Desprez, F., Jeannot, E., Jégou, Y., Lanteri, S., Leduc, J., Melab, N., Mornet, G., Namyst, R., Primet, P., Quetier, B., Richard, O., Talbi, E.G., Touche, I.: Grid'5000: A large scale and highly reconfigurable experimental grid testbed. International Journal of High Performance Computing Applications 20(4), 481–494 (Winter 2006)

[8] Candan, K., Tatemura, J., Agrawal, D., Cavendish, D.: On overlay schemes to support point-in-range queries for scalable grid resource discovery. Fifth IEEE International Conference on Peer-to-Peer Computing (P2P'05) (2005)

[9] Fernandez, H., Priol, T., Tedeschi, C.: Decentralized Approach for Execution of Composite Web Services Using the Chemical Paradigm. In: 17th IEEE International Conference on Web Services (2010)

[10] Gladitz, K., Kuchen, H.: Shared memory implementation of the Gamma-operation. Journal of Symbolic Computation 21(4-6) (1996)

[11] Grumbach, S., Wang, F.: Netlog, a rule-based language for distributed programming. In: 12th International Symposium on Practical Aspects of Declarative Languages. pp. 88–103 (2010)

[12] Huang, L., Tong, W., Kam, W., Sun, Y.: Implementation of GAMMA on a Massively Parallel Computer. Journal of Computer Science and Technology 12 (1997)

[13] Laliwala, Z., Khosla, R., Majumdar, P., Chaudhary, S.: Semantic and rules based Event-Driven dynamic web services composition for automation of business processes. In: Services Computing Workshops, 2006. SCW '06. IEEE. pp. 175–182 (2006)

[14] Lin, H., Kemp, J., Gilbert, P.: Computing Gamma Calculus on Computer Cluster. International Journal of Training and Development 1(4) (2010)

[15] Lloyd, J.W.: Practical advtanages of declarative programming. In: Joint Conference on Declarative Programming (GULP-PRODE'94). pp. 18–30 (1994)

[16] Milojicic, D., Kalogeraki, V., Lukose, R., Nagaraja, K., Pruyne, J., Richard, B., Rollins, S., Xu, Z.: Peer-to-peer computing (2002)

[17] Mostéfaoui, A.: Towards a computing model for open distributed systems. In: 9th International Conference on Parallel Computing Technologies (PaCT). pp. 74–79 (2007)

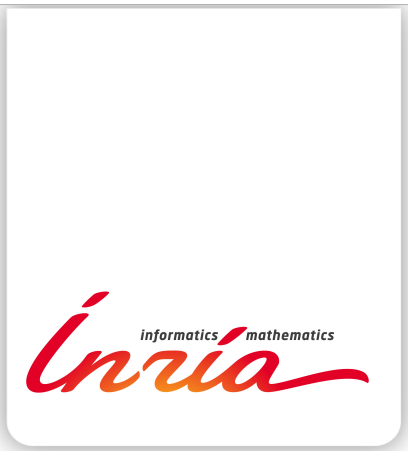
[18] Napoli, C.D., Giordano, M., Pazat, J.L., Wang, C.: A chemical based middleware for workflow instantiation and execution. In: ServiceWave. pp. 100–111 (2010)

[19] Obrovac, M., Tedeschi, C.: When Distributed Hash Tables Meet Chemical Programming for Autonomic Computing. In: 15th International Workshop on Nature Inspired Distributed Computing (NIDisC 2012), in conjunction with IPDPS 2012. IEEE, Shanghai, China (May, 21 2012), to appear

[20] Rowstron, A., Druschel, P.: Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. LNCS 2218, 329–350 (2001)

[21] Schmidt, C., Parashar, M.: Squid: Enabling search in DHT-based systems. Journal of Parallel and Distributed Computing 68(7) (2008)

[22] Wang, Y., Li, M., Cao, J., Tang, F., Chen, L., Cao, L.: An ECA-Rule-Based workflow management approach for web services composition. In 4th Int. Conference on Grid and Cooperative Computing (GCC)'05 3795, 143–148 (2005)



RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399