Distributed Chemically-Inspired Runtimes for Large Scale Adaptive Computing Platforms

Cédric Tedeschi

► **To cite this version:**

Cédric Tedeschi. Distributed Chemically-Inspired Runtimes for Large Scale Adaptive Computing Platforms. Distributed, Parallel, and Cluster Computing [cs.DC]. Université de Rennes 1, 2017. tel-01665776

HAL Id: tel-01665776

https://theses.hal.science/tel-01665776

Submitted on 16 Dec 2017

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.

N^o d'ordre : ANNÉE 2017

HABILITATION À DIRIGER DES RECHERCHES

UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

Mention Informatique
École doctorale Matisse

présentée par

Cédric Tedeschi

préparée à l'unité de recherche IRISA (UMR 6074)
Composante universitaire: ISTIC

Distributed Chemically-Inspired Runtimes for Large Scale Adaptive Computing Platforms

HDR soutenue le 11 avril 2017 à Rennes
devant le jury composé de :

Claudia Di Napoli
Chargée de recherche / CNR / Rapporteuse

Jean-Louis Giavitto
Directeur de recherche / CNRS / Rapporteur

Jean-Marc Jézéquel
Professeur / U. de Rennes 1 / Président

Johan Montagnat
Directeur de recherche / CNRS / Examinateur

Christine Morin
Directrice de recherche / Inria / Examinatrice

Thierry Priol
Directeur de recherche / Inria / Examinateur

Pierre Sens
Professeur / U. Pierre et Marie Curie / Rapporteur

Contents

Introduction **5**

1 The Chemical Programming Style **13**
1.1 Chemically-inspired programming . . . 13
1.2 HOCL . . . 14
1.3 Close models . . . 15

2 On the Runtime Complexity of Concurrent Multiset Rewriting **17**
2.1 The complexity problem . . . 17
2.2 Model . . . 18
2.2.1 Modeling of the rule's condition . . . 18
2.2.2 Structuring of the multiset . . . 19
2.2.3 NP-hardness and the subgraph isomorphism problem . . . 20
2.3 Calibre and the PMJA Algorithm . . . 21
2.3.1 Calibre of a rule . . . 21
2.3.2 The PMJA algorithm . . . 23
2.4 Discussion . . . 26
2.4.1 Summary . . . 26
2.4.2 Approximation and distribution . . . 26

3 Adaptive Atomic Capture of Multiple Molecules **29**
3.1 The importance of being atomic . . . 29
3.2 System model . . . 31
3.2.1 Data and Rules Dissemination . . . 31
3.2.2 Discovery Protocol . . . 31
3.2.3 Fault tolerance . . . 32
3.3 Protocol . . . 32
3.3.1 Pessimistic sub-protocol . . . 32
3.3.2 Optimistic sub-protocol . . . 33
3.3.3 Sub-protocol mixing . . . 35
3.3.4 Dormant nodes . . . 36
3.3.5 Multiple-rules programs . . . 36
3.3.6 Protocol's safety and liveness . . . 37
3.4 Evaluation . . . 38
3.4.1 Single-rule experiments . . . 39
3.4.2 Multiple-rules experiments . . . 42
3.5 Conclusion . . . 43

4 Implementing Distributed Chemical Machines **45**
4.1 The need for chemical run-time environments . . . 45
4.1.1 Physical parallelism . . . 46

4.1.2 Physical distribution . . . 46
4.2 A hierarchical reactor . . . 46
4.2.1 Overview . . . 47
4.2.2 Efficient condition checking and inertia detection . . . 47
4.2.3 Tree reorganisation . . . 49
4.2.4 Prototype . . . 50
4.2.5 Experimental evaluation . . . 51
4.3 A fully-decentralised higher-order reactor . . . 53
4.3.1 General execution scheme . . . 54
4.3.2 Molecule searching and inertia detection . . . 55
4.3.3 Higher-order . . . 56
4.3.4 Prototype . . . 58
4.3.5 Experimental evaluation . . . 59
4.4 Conclusion . . . 61

5 Decentralising Workflow Execution 63
5.1 Workflows, workflow managers, and decentralisation . . . 63
5.2 Shared-space based coordination . . . 64
5.3 Chemical workflow specification . . . 65
5.4 GinFlow . . . 67
5.4.1 The HOCL distributed engine . . . 68
5.4.2 Resilience . . . 69
5.4.3 The executors . . . 69
5.4.4 Client interfaces . . . 70
5.5 Experimental validation . . . 70
5.5.1 Performance . . . 72
5.5.2 Impact of the executor and messaging middleware . . . 72
5.5.3 Resilience . . . 73
5.6 Decentralising workflow execution in literature . . . 73
5.7 Conclusion . . . 74

6 Injecting Adaptiveness in (Decentralised) Workflow Execution 77
6.1 The need for dynamically adapting workflows . . . 77
6.1.1 *Cold* adaptation . . . 77
6.1.2 *Hot* adaptation . . . 78
6.2 HOCL-based adaptive workflows . . . 79
6.2.1 HOCL abstract adaptive workflow . . . 79
6.2.2 Adaptation rules . . . 80
6.2.3 Execution . . . 80
6.2.4 Generalising . . . 81
6.3 Adaptiveness in GinFlow . . . 82
6.3.1 Implementation . . . 82
6.3.2 Client interfaces . . . 82
6.4 Experimental evaluation . . . 83
6.5 Adapting the workflow shape in literature . . . 84
6.6 Conclusion . . . 85

Conclusion 87

Bibliography 91

Introduction

The quest for distributed programming styles

Every time a new type of computing platforms appears, the challenge comes back: increasing the amount of computing resources does not guarantee that such a computing power will be adequately exploited. There is a crucial need to offer programming tools, styles, abstractions that will allow the user to easily specify and efficiently run programs over the platform.

The difficulty stands in the self-contradiction conveyed by such a venture: *ease* most of the time means raise the level of abstraction, while being efficient requires sometimes to have a manual control over the details of the platform's architecture. The quest for the *right* level of abstraction can be traced back to the foundational crisis traversed by mathematicians in the early twentieth century, and which saw the rise of the field of computability, which was trying to formally decide what can be computed and what cannot. At that time, the question of the programming *style* was not the primary concern.

In his 1936 seminal paper about computability [100], Alan Turing describes first his abstract machines in the most basic way possible, to avoid any misunderstanding. In particular, a machine includes a very limited *instruction set*: move to the next symbol left or right, read one symbol on the tape, change its inner state. Yet, at some point, and for the sake of describing his machines in a more concise way, Turing introduces what he refers to as an *abbreviation*: the *skeleton tables*, which allows to describe more easily machines, with generic functionalities that can take parameters. The key point is that, as he states [100]:

> *The skeleton tables are to be regarded as nothing but abbreviations: they are not essential. So long as the reader understands how to obtain the complete tables from the skeleton tables, there is no need to give any exact definitions in this connection.*

While Turing presents the skeleton tables as not *essential* (in the sense that it will not change anything in terms of computability), it is still a small step towards raising the level of abstraction, towards more *expressiveness*.

A few years later, two main families of programming languages emerged. One was based on what is now referred to as the Von Neumann architecture model of a computer [102], which, while simple, offered a model encompassing most computer architectures in those days of rapidly changing technology. The Von Neumann model paved the way for the family of imperative languages. The second family of programming style took its roots in Lambda-calculus, developed by Alonzo Church in an attempt at building a formal ground to computability and generally considered equivalent to other mathematical model of computability such as partial recursive functions [34]. While less correlated to the physical architecture of a digital computer, lambda-calculus was the ground the other family of programming languages, called functional programming.

To put it simply, while imperative programming requires the programmer to deal with explicit machinery instructions such as reading and writing memory cells and incrementing pointers, functional programming focus on building a functional solution to a problem, *i.e.,* a solution involving merely the evaluation of a composition of functions. While the passionate debate about the pros and cons of these two styles is not the primary concern of the present dissertation, the point is that the functional style abstracts out the details of the underlying architecture. This is not true in the case of imperative programming which explicitly deal with memory manipulation, even if some imperative languages, such as C, offers an increased level of abstractions when compared with assembly languages.

More generally, functional programming is to be considered as a *declarative* paradigm. Declarative programming [68] tries to separates the logic of a computation (*"what we want to do"*) from its control (*"how to achieve it"*). More precisely, in such an approach, while the "what" is to be defined by the programmer, the "how" is ideally left to the runtime system, the engine of the execution environment. Unfortunately, increasing the level of abstraction comes at a price: the more declarative (or *expressive*) a language is, the more difficult it becomes to have it executed, in average, in an efficient way.

This picture can be extended to distributed computing. Even if of a higher-level of abstraction, message passing can be thought of as the *assembly language of distributed systems*: one computer sending bits to another one through a channel. Message passing does not constitute a programming model in itself, in the sense that it was not developed to ease the programmer's specification work. Remote Procedure Call (RPC) is a programming model allowing to call procedures to be run on distant computing facilities from within a local program. It operates in a client-server fashion: the client, which is running a program locally, relies on some remote procedure for parts of its execution. When this remote procedure needs to be called, the client sends the parameters on which to apply the function. The server receives them, runs the function on them, and sends the result of the execution back to the client, that can now use it for subsequent local computation steps. RPC was also developed as an extension of the object-oriented model, as proposed by RMI [42] in the Java world or by CORBA [92]. The principles of RMI and CORBA are similar to those of RPC except that functions called are methods from objects, these objects being hosted by distinct computers, or at least by distinct Java virtual machines in the case of RMI.

CORBA and similar mechanisms were used in the development of Grid computing, in which the computing power of heterogeneous geographically distributed computing facilities (clusters or desktops) is aggregated into a single entity: the *grid*. Taking into account the extra requirements related to the scale, the heterogeneity and the unreliable nature of large scale platforms, led to the rise of the component model [98].

Concretely speaking, the component model [98] and its different flavours and implementations [15, 24, 14] were developed to answer to the need to compose existing pieces of stand-alone software. As for the Object-Oriented programming style, reuse and ease of composition were the main drivers behind the development of the component model. However, new needs were also to be tackled, mainly heterogeneity and distribution: the software services to be composed can rely on different languages and run on different computing facilities. Yet, we need a way to encapsulate them so as to make them communicate transparently. A common case for components is code coupling, as explained in the paper by Denis *et. al* [37]: the authors describe a simple code coupling-based application that simulates the transport of chemical products in a porous medium in the context of a chemical 3D simulation. In this example, two codes have been developed separately by different teams in different languages: a first code computes the chemical product's density and a second one simulates the medium's porosity. These two codes typically run on different distant computing facilities. To run the whole simulation, an appealing solution would be to combine these two codes (instead of trying to build another code from scratch. Still, we have to solve the heterogeneity and communication issues. In our example, two components are to be deployed. Each service will be encapsulated into a component able to communicate with the other, following the *use and provide* paradigm, each component having a set of ports of two possible kinds: ports providing data to others, called *facets*, and ports receiving data from others, called *receptacles*.

The Internet of Services

The last shift the world witnessed in terms of digital ecosystem led to the rise of a global platform delivering a virtually unlimited computing power including on one side very large centralised infrastructures, resulting from the morphing of computing clusters into utility computing centers (or *clouds*), and on the other side a myriad of less heavy devices such as smartphones, tablets and TVs. This global digital ecosystem revolves around the Internet backbone and supports a boundless number of computing *services* ranging from all possible brands of on-line business services to scientific data storing and processing services helping scientists with their researches.

In the context of the Internet of Services, the term *service* encompasses a large set of functionalities, implemented by various languages, supported by heterogeneous software stacks and systems running on

different hardware. Still, they tend to have a set of common characteristics. Let us try to define what a *service* is. I think we can safely state three basic characteristics a service must have: i) an unambiguous self-contained functionality, ii) a network-enabled, well-defined API, iii) the ability to be composed with other services.

The last point raises a lot of questions related to the programming abstractions, concepts, architectures and tools that could enact a service composition in such an environment. Interconnecting computing services has been a major topic in both industrial and academic environments since the rise of the dot-com era in the middle of the 1990s. Component architectures and workflows have appeared to be the two major paradigms to handle this problem.

Components *vs* workflows

The limitation of the component models stands primarily in that these ports mentioned above are specific to an application and can only be statically defined. In other words, a component is compatible only with a predefined set of other components which have been constrained with a symmetric set of ports, in order to build the targeted application. At run time, the components cooperate in a remote procedure call fashion. This can be seen as *spatial composition*: services are all present at the same time, and these service boxes are interconnected in the *application plane*, so that they can call each others. Spatial composition relies on the encapsulation of the services to be composed: the details of the bindings between components are developed inside the components, through stubs and skeletons acting as proxies to the other components. Unfortunately, this static encapsulation hinders the possibility for one service to communicate with services other than those predefined for this specific application's purpose.

The other kind of composition, still as described in [22], is *temporal composition*. By *temporal*, it is meant that this kind of composition expresses the actual control flow between the set of services involved: it focus on *when* services need each others, in other words, what are the data and control dependencies between them. This paradigm is also known as distributed *workflow* computing. A workflow is most of the time a directed acyclic graph where each vertex is a task and each edge is a dependency between two tasks, either a control dependency or a data dependency.

In workflow computing, composed services are not really encapsulated: the composition of the services itself is described in a distinct file, to be read and enacted by a *workflow engine*, or *service orchestrator*, which is external to the services. The services do not know themselves how to take part in the execution and its coordination (*i.e.,* the mechanisms enforcing the satisfaction of the dependencies expressed in the workflow). Basically, a service does nothing except waiting for a call. This is where the *orchestrator* is brought into the picture. The orchestrator enacts the workflow described in the description file. At run time, it takes care of the coordination by triggering the tasks in the order specified. It acts as a central coordinator. Every notification of the completion of a service, or even intermediate data produced will go through it.

We generally distinguish two kinds of workflow: *abstract* ones and *concrete* ones. An abstract workflow is typically a DAG of tasks where tasks represent *what* is to be done. This is the functional specification of the workflow. A fully abstract workflow does not include any technical information such as how to enact these tasks or what specific executable file will actually get called to realise each task. On the other side of the spectrum, a concrete workflow contains all the technical details needed to actually execute the workflow and answers questions such as *where are the data?*, *where is the executable file to be used?*, *what computing nodes will host the execution of what task?*

This discussion leads us to Table 0.1 which summarizes the differences between component-based applications and workflows. The first dimension, the *type* of composition, is the essence of each paradigm. It has been discussed above and in some way subsumes the two following dimensions: *binding* and *coupling*.

The second dimension in Table 0.1, binding, refers to *when* the communication between two services taking part in the workflow is actually established. In workflows, it can be done *just-in-time*. It is useless to bind any service to perform some task before its incoming dependencies are satisfied. Late binding brings flexibility to the execution as it allows to choose the best service available on the platform when the enactment

	composition	**binding**	**coupling**	**coordination**	**adaptation**
components	spatial	static	tight	decentralised	static
workflows	temporal	dynamic	loose	*centralised*	*static*

Table 0.1: Components *vs* workflows.

of the tasks is really needed. In component architecture, this is not possible, as binding information needs to be included within components at design time.

The third dimension, *coupling*, refers to the degree of freedom between services. Typically, a component is developed so it can communicate with few other, well-identified components and only those. In other words, components are *tightly-coupled*. As stated before, when they are not encapsulated in components, services are free of any by-design coupling. They just provide an API so any other service can actually *talk* with them when needed (provided it respects the API.)

In component architectures, coordination between components is specified at design time following the RPC programming model, and is thus distributed, each component being able to *call* other components without relying on any external third-party orchestrator. In contrast, workflow managers are traditionally based on a centralised orchestrator. This brings the common drawbacks of centralisation: poor resilience and a limited scalability. However, decentralising workflow coordination brings difficulties. In decentralised settings, each service is responsible for indicating to others when to start, and should be able to receive the same kind of notification. As we do not want to lose the late binding and loose coupling properties, services need to be able to share and update information about the status of the workflow so as to execute it without relying on a central orchestrator. This is one major topic developed in the following of this document:

how to decentralise the coordination of workflow execution?

The final dimension is *adaptation*. In a general sense, adaptation refers to the ability to modify some behaviour when needed. The need for adaptation generally comes after a change or a failure in the infrastructure. It may also come from the workflow itself, whose execution (partial) outcome does not give satisfaction to its designer. While the first case can be solved by traditional fault-tolerance mechanisms, the second case, which deals with the workflow composition and structure itself, requires to provide the programmers with tools allowing them to deal with dynamic adaptation of the workflow composition and structure and its (on-the-fly) re-deployment. This need is particularly present in many science domains, where the design of a workflow resembles an exploration. As put by Gil. *et al.*, in [52]:

> ... *most scientific activity consists of exploration of variants and experimentation with alternative settings, which would involve modifying workflows to understand their effects and how to explain those effects. Hence, an important challenge in science is representation of workflow variants, which aims at understanding the impact that a change has on the resulting data products as an aid to scientific discourse.*

There is a need to provide a way for the programmer to tag portions of the workflow to be supervised along with alternate sub-workflows to replace them in case they do not give satisfying outputs. Here, *satisfying* depends strongly on the domain considered. (Only specific scientists can decide on the relevance of intermediate data obtained.) Then, given these elements, the runtime system should be able to modify the structure of the current workflow so as to go from the initial configuration to the alternate one. Dynamic adaptation has been largely ignored in the design of workflow managers. This constitutes a second major topic discussed in this dissertation:

how to adapt the workflow structure on-the-fly (in decentralised settings) ?

Performing this adaptation at run time, with no human intervention, and moreover in decentralized settings calls for techniques belonging, broadly speaking, to the field of autonomic computing.

Autonomic Computing

As stated before, as systems get more complicated, alternate programming models need to be proposed. More precisely, above a certain level of complexity, a top-down approach for specification may appear necessary: the system's specification is refined step by step starting from a high level view of how the system is supposed to behave. In other words, it supposes having several levels of specifications for the same system, as the whole system cannot be specified in detail at once using a unique language. Also, facing such a complexity means that human intervention at run time is not desirable and simply not feasible. Systems must be able to manage themselves, they need to become *autonomic*. As written in IBM's whitepaper on autonomic computing [57]:

> *Autonomic Computing helps to address complexity by using technology to manage technology. The term autonomic is derived from human biology. The autonomic nervous system monitors your heartbeat, checks your blood sugar level and keeps your body temperature close to 98.6 °F without any conscious effort on your part. In much the same way, self-managing autonomic capabilities anticipate IT system requirements and resolve problems with minimal human intervention.*

When autonomic computing gained *momentum* in the early 2000s, it echoed the pioneering works in cybernetics conducted fifty years earlier. In the 1950s, nature was already a source of inspiration, and in particular the human brain, as developed by William Ross Ashby in his book *Design for a Brain* [6]. The analogy between a digital autonomic system and the autonomous nervous system was exposed through an example of *homeostasis*, namely glucose regulation:

> ... *if the concentration should fall below about 0.06 percent, the tissues will be starved of their chief source of energy; if the concentration should rise above about 0.18 percent, other undesirable effects will occur. If the blood-glucose concentration falls below about 0.07 percent, the adrenal glands secrete adrenaline, which causes the liver to turn its stores of glycogen into glucose. This passes into the blood and the blood glucose concentration drop is opposed. Further, a falling blood-glucose also removed by muscles and skin, and if the blood-glucose concentration exceeds 0.18 percent, the kidneys excrete excess glucose* ...

Let us linger a bit on this description: It mixes two questions: *what are the threshold values to start regulating?* and *how the regulation is achieved?* Actually, it seems that the *what to do* could be simplified by the two following rules (let c the blood glucose concentration):

$$\begin{cases} c < 0.07\% \rightarrow & \text{secrete adrenalin} \\ c > 0.15\% \rightarrow & \text{secrete insulin} \end{cases}$$

These two simple lines may constitute the high-level rules the brain needs to adhere to in order to maintain glucose blood concentration in survival areas: it is the *specification.* To actually achieve the constraints expressed by the rules, it relies on complex chemistry taking place within the body: it is the *implementation.* Autonomic systems advocate the separation of concerns between specification and implementation, which should not be addressed at once. As stated by Parashar and Hariri in [80] in 2004:

> *Conceptual research issues and challenges include defining appropriate abstractions and models for specifying, understanding, controlling, and implementing autonomic behaviors.*

The need for adequate programming models, in particular for specification purposes, has been raised once again with the rise of autonomic computing.

Chemically-inspired Programming

So what? What programming style should be used to bring decentralization and adaptiveness to large scale (workflow) coordination? The old debate between imperative and declarative programming is still highly relevant when it comes to separate specification from implementation in large distributed systems. In a 2010 paper devising a rule-based language to specify distributed applications such as communication protocols, Grumbach and Wang wrote [54]:

> *Rule-based languages allow to obtain code about two orders of magnitude more concise than standard imperative programming languages.*

Rule-based languages are a typical example of a declarative language where the *what* is explicitly stated through a set of rules to be satisfied. This is why it has been extensively used in expert systems [55]. While rule-based programming styles are appealing in our case, we still need one which can easily capture the distributed nature of the coordination while offering ways to express dynamic adaptation. This is where the programming style brought about by artificial chemistries comes into the picture [39].

The chemically-inspired programming style was initially proposed with the objective of offering a declarative programming style for highly parallel programs. In a 1988 paper [9], the authors propose a chemically-inspired programming style based on multiset rewriting. They establish the absence of structure in data and of sequentiality in control as the norm. And because no data structuring and plain parallelism is the rule, they can become implicit. In the end, the programming style proposed can be summarized as a set of rules to be applied concurrently on some input unstructured data, more formally a multiset.

At design time, the programmer *just* needs to define the set of rules to be applied on the multiset, and specifies the initial content of the multiset (in more traditional terms, input data). The programmer assumes that, at run time, these rules are to be applied in no specific order on the multiset. There is only one assumption: if one rule can be applied given the state of the data, one rule will be applied in a finite time. It is the duty of the runtime implementor to make the relevant choices to implement this relative freedom efficiently.

The higher order in chemical programming was introduced in 2007 [11], through the proposal of the Higher-Order Chemical Language (HOCL). In HOCL, rules can be applied to other rules. In other words, rules can appear or disappear from the program depending on the state of the multiset. This paves the way for dynamic adaptation when using a rule-based programming style.

Chemical computing at the rescue; At the rescue of chemical computing

This dissertation presents a series a works conducted between 2009 and 2016 pursuing the idea that higher-order chemical programming is a relevant programming style for the autonomous execution of applications at large scale. Still, until recently, a significant barrier towards the actual adoption of the chemical style stood in the lack of runtime systems supporting it, especially at large scale. This is another major topic developed in this dissertation:

what are the concepts, algorithmics and software pilots needed towards a largely distributed runtime system supporting the chemical programming style ?

Outline

The first part of this dissertation presents the results obtained on this specific research tracks. Chapter 1 presents the chemical programming model in more detail. Chapter 2 discusses its complexity at run time. Chapter 3 goes large scale and addresses the algorithmic challenges raised by distributing the runtime of chemical programs. Chapter 4 presents the design, implementation and experimental validation of actual *distributed chemical machines*.

The second part of this dissertation builds upon the idea that the chemical model, as a representative of concurrent rule-based programming eases the design of decentralised and adaptive coordination models and environments for workflow execution. Chapter 5 describes the design, implementation and experimental validation of *GinFlow*, a decentralised workflow executor whose execution agents rely on a chemical engine at their core. Chapter 6 shows how such a decentralised workflow execution can be enhanced with adaptiveness at the workflow level to provide workflow developers with a flexible design tool where different workflow alternatives can be designed and enacted dynamically.

Chapter 1

The Chemical Programming Style

In this chapter, we discuss the chemical programming style and present the Higher-Order Chemical Language (HOCL), the language chosen to underlie our work on workflow execution and adaptation and prototype the GinFlow software, as presented later in Chapters 5 and 6. We finally mention few close models from the literature.

1.1 Chemically-inspired programming

Chemical programming was born from the idea that trying to bring sequential programming models to the parallel world is not just a matter of extension. Approaches such as Open-MP [82], in which parallelizable parts of a sequential program can be delimited by the programmer through the insertion of *pragmas*, cannot be an effective solution if the program can be made massively parallel. In this case, one appealing option is to change the semantics of the order of instructions. In imperative programs, sequentiality does not need to be made explicit: the order of statements settles it. The idea followed by chemical programming is to change this fundamental semantic rule: the order in which instructions are written no longer matters. They will be parallel by default.

Like other declarative programming style, chemical programming allows the programmer to concentrate on the very essence of the algorithmics needed to solve a given problem. It takes after the idea of minimalism: in particular, when no control or data structuring is needed, then no structuring will be used. Note that structuring often comes from the fact that things has to be made sequential. This is particularly striking with imperative language. Let us consider the problem of extracting the highest value among a set of integers. Using a programming language such as C, one will end up with a function somehow resembling the code below:

```
int getMax (int tab [], int size) {
    int max = tab[0];
    for (int i = 1; i < size; i++) {
         if (tab[i] > max)
              max = tab[i];
    }
    return(max);
}
```

In this example, the structuring of the data (a set of integers) into a table comes from the need to read the elements one by one. A table is not necessary to solve the problem, whose only inherent structuring (or absence thereof) is embodied in the set, or *multiset* if some integers can appear more than once. *GetMax* is a problem whose solving can be made highly parallel, due to its associative nature. The very essence of the algorithmics behind this problem is comparison, and these comparisons can be done in any order. In fact, we can express the processing needed as a single (HOCL) rule (let us call this rule *max*):

$$\textbf{let } max = \textbf{ replace } x, y \textbf{ by } x \textbf{ if } x \geq y \tag{1.1}$$

The rule selects two elements from the multiset and keeps only the one with the highest value. It captures the fundamental concepts behind solving the *getMax* problem. If the rule is applied a sufficient number of times, the multiset will necessarily end up containing a single integer, having the highest value of the initial multiset of integers provided. The *max* rule is actually a valid rule of a chemical program. Let us complete it with some values in the initial multiset, or *chemical solution* (described after the **in** keyword and delimited by $\langle$ and $\rangle$). We obtain the following program:

$$\textbf{let } max = \textbf{ replace } x, y \textbf{ by } x \textbf{ if } x \geq y \textbf{ in } \langle 2, 3, 9, 5, 8 \rangle$$

At run time, the *engine* responsible for its execution will try to apply the rule on the multiset, *i.e.,* find molecules in the solution satisfying the pattern and condition of the rule. Note that the pair in the pattern is ordered and that the first element of the ordered pair found needs to be at least equal to the second element in order for the rule to be applied. Of course, any pair of integers works if we are able to test its two possible orders, or if a *maximum* function is available, in which case Rule 1.1 can be reformulated as following:

$$\textbf{let } max = \textbf{ replace } x, y \textbf{ by } maximum(x, y)$$

Initially, several reactions are possible in the provided solution: between 2 and 3, 2 and 5, 8 and 9, *etc.* At run time, the rule will be applied in some order (unknown, left to the interpreter's developer). Whatever the order is, the final content of the multiset will be $\langle 9 \rangle$. In other words, one possible evolution of the multiset during execution is the following:

$$\langle 2, 3, 9, 5, 8 \rangle \rightarrow^* \langle 3, 9, 8 \rangle \rightarrow^* \langle 9 \rangle$$

$\rightarrow^*$ denotes the fact that several reactions took place (in an unknown order). What we can infer from this fragmented observation is that 2 and 5 disappeared first (and we do not know what was their counterpart in the reaction) and that, in a second phase, 3 and 8 disappeared. What we know for sure is that the last reaction involved 9.

Summing up, the execution model assumes that whatever the number of rules of one program is, these rules are to be applied concurrently in no particular order on the multiset, until no reactions can be applied anymore, *i.e.,* until no subset of elements satisfying any of the reaction rules' condition can be found in the solution. The program is then said to be *inert*. In this state, the solution will not change anymore and contains the final result of the program. There are however, two limitations to this concurrency: (1) The application of some rule cannot be delayed indefinitely if some rule is enacted. (2) A molecule cannot take part in more than one reaction during its lifetime. You may have noticed that molecules are not mutable. They are produced in a reaction, and then possibly consumed in another reaction. It means that, if several rules are to be applied at the same time, they cannot share a part of their *reactants*. This assumption called *atomic capture*, is crucial for the correctness of the programs. In distributed settings, enforcing it brings about interesting problems that we tackle in Chapter 3.

To make our program a valid HOCL program, we actually need to add the rule into the multiset, due to the fact that HOCL provides the higher order: rules are first-class citizens in the multiset. In fact, *max* must be present in the solution from the beginning to the end of the execution, as a rule can be applied only if it actually appears in the solution. We only omitted it for the sake of clarity.

1.2 HOCL

The Higher Order Chemical Language (HOCL) was proposed in 2006 as a step towards more expressiveness and usability of the chemical programming style [11]. The main extra feature of HOCL added to previous chemical programming languages such as Gamma [8] is the higher-order: In HOCL, rules are first class citizens in the solution, and a rule can apply on other rules. Consequently, rules can appear and disappear from the solution, enabling or disabling them dynamically. Let us illustrate the higher order on our running

example. As mentioned before, the *max* rule remains in the solution at the end of the execution. Removing it (so as to only keep the greatest integer) calls for higher-order mechanisms, in our case, an extra *clean* rule consuming the *max* rule. A problem is that *clean* should be enabled only when *max* rule is no longer necessary, which is not the case by default. If one rule is present in the multiset, it may be triggered. Envisioned differently, we need to inject a bit of sequentiality into the execution: first, apply *max* until it cannot react anymore, and secondly apply *clean* to remove *max*. This is achieved through *subsolutioning*. Another problem is that this new rule must be removed in its turn. This can be achieved by specifying that the *clean* rule can be triggered only once. This leads to the following program:

$$\begin{aligned}&\textbf{let } max = \textbf{ replace } x, y \textbf{ by } x \textbf{ if } x \geq y \textbf{ in}\\&\textbf{let } clean = \textbf{ replace-one } \langle max, \omega \rangle \textbf{ by } \omega \textbf{ in}\\&\langle \langle 2, 3, 5, 8, 9, max \rangle, clean \rangle\end{aligned}$$

The program has been restructured to encapsulate our initial program in an outer multiset containing the *clean* rule which extracts the result from the inner multiset, and removes *max* at the same time. The ω symbol has some special semantics: it can match any multiset. In our case, it will match the molecule remaining after the repeated application of the *max* rule within the subsolution. HOCL ensures, by definition, that the content of a subsolution cannot react as long as this subsolution is not inert. Consequently, the *clean* rule cannot react as long as the subsolution containing the *max* rule is not inert, and, in our specific example, the 9 is extracted and the *max* rule is removed. Finally, note that the *clean* rule is a **replace-one** rule: It is one-shot and will disappear from the multiset once triggered, leaving the solution as $\langle 9 \rangle$.

This example shows how a program change dynamically through the injection or removal of some rules, paving the way for online reconfiguration. It also suggests that the multiset is a container for the state of the program, on which an engine (or multiple engines) can apply rules (concurrently).

To complete this overview of HOCL, let us give a summary of what we can find within an HOCL solution. The solution is a multiset of atoms. An atom can be either a simple one such as a number, a string, and a rule, or a structured one. A structured atom can be either a subsolution (a multiset inside the multiset), denoted $\langle A_1, A_2, \ldots, A_n \rangle$ or a tuple (an ordered multiset) denoted $A_1 : A_2 : \cdots : A_n$. HOCL can also use external functions that, given a set of atoms as input will return another set of atoms. For instance, the HOCL interpreter used within GinFlow (the subject of Chapter 5) can call Java methods.

1.3 Close models

Since the pioneering work [9] that led to the development of Gamma [8], several series of works developed the chemical style in different directions. The CHAM (CHemical Abstract Machine) model formalised the chemical programming style while being able to model locality: Solutions can be nested (within membranes) and evolve independently [16]. Also, the CHAM introduces *airlocks* allowing to move molecules from one membrane to another one. Pushing the chemical metaphor, the CHAM includes particular rules to *heat* molecules so as to break complex ones into their components and to *cool* a set of components to rebuild the compound molecule from them. In the same vein, P-systems propose a programming model based on nested independently evolving membranes (or *cells* by analogy with biology) [81]. P-systems can have rules expressing molecules to move in and out a cell, and membranes to get constructed or dissolved. One of the scheduling model proposed in P-Systems consists in a discrete time execution where at each step, the set of rules applied is the one that consumes the largest possible amount of molecules. Note that the atomic capture is still required. However, other models of parallelism have been proposed in the literature of P-systems [81].

Many works went into exploring artificial chemistries (ACs), building similar models, but in an attempt to model real chemical processes (just as an artificial life tries to model real life mechanisms). Generally speaking, an AC is fully characterised by (i) its molecules, (ii) its possible reactions, and (iii) the way they appear in time *i.e.*, how its dynamics are modelled [39]. These three components are indeed influenced by the goal of an AC. To take an example, some simple ACs were developed to study the emergence and evolution of a metabolism [7]. In this case, molecules are simple strings, in which characters are taken

from a finite alphabet representing the possible monomers composing the molecules. Possible reactions are 1) the concatenation and 2) the cleavage of such strings. These reactions appear as in a well-stirred tank, *i.e.,* the dynamics consist in generating a set of realistic sequences of reactions. As studying all the possible sequences of reactions is intractable, some meta-dynamics are introduced to limit the number of species present in the solution and their concentration at a given point in time, in turn limiting the possible sequences. While such an AC is relevant for metabolism modelling, it cannot be used to model concurrent computations due to its restrictive meta-dynamics. In the same vein, let us mention the work of Suzuki and Tanaka about the emergence of cycles in chemical systems through the random generation of rewriting rules [97]. Fontana defined a rewriting-based AC built on top of λ-calculus, and aimed at giving a minimal formalisation of biological organisation : in this system, two λ-expressions can react together to produce a new λ-expression [49].

The MGS language is dedicated to the simulation of biological processes and offers a unification of several models inspired by biology and chemistry (Gamma, CHAM, P-systems, Lindenmayer systems and cellular automata). In MGS, the basic operation is the local transformation: replacing a sub-collection of elements by another one in a global collection. MGS includes the possibility to express topological information about the elements in a collection, so as to constrain transformations on topologically related elements (elements related in some way through a neighbourhood relationship) [51].

Besides HOCL, the higher order has been introduced independently in [65] and [31], but none of these works fully integrate the higher order, neither conceptually (programs and data are still separated in [65]) nor practically (the goal of the calculus presented in [31] is not to build an actual programming language).

Let us finally mention that all these models share similarities with Petri Nets (PNs) in the sense that PNs can be seen as multiset rewriting models: Places are multisets of tokens and transitions are the applications of some rules on these multisets, producing new tokens to appear in destination places. If you add types and conditions to these multisets of tokens, you obtain coloured PNs [60].

Chapter 2

On the Runtime Complexity of Concurrent Multiset Rewriting

In this chapter, we study the runtime complexity of the chemical programming model and characterise it using an analogy with the subgraph isomorphism problem.

Joint work with:

- ⋆ Matthieu Perrin (Master student, ENS de Rennes)
- ⋆ Marin Bertier (Associate professor, INSA de Rennes)

2.1 The complexity problem

While the chemical programming model is envisioned as a promising way to specify autonomic systems, one of the main barrier towards its actual adoption is related to its execution complexity : each computation step (*i.e.*, the application of one rewriting rule) assumes that some *reactants* satisfying the rule's condition are found in the multiset. Let us assume the number of objects in the multiset is n, and the *arity* of the rule, (*i.e.*, the number of reactants needed for its application) is k. Then, in the worst case, an exhaustive exploration of all possible combinations of k molecules among n is needed, and the complexity involved is in $O(n^k)$ (assuming $n \gg k$), which makes it, when k increases, an intractable problem. One question left largely open about the model is the possibility to improve the time of reactants search (at least in some cases to be identified). The present work targets chemical models based on multiset processing allowing to have complex conditions on rules, such as [8, 11, 31].

In this chapter, we explore the possibility of improving the complexity of searching reactants through a static analysis of the reaction condition. In particular, we give a characterisation of this complexity, by analogy to the subgraph isomorphism problem. Given a rule R, we define the *rank* $\mathrm{rk}(R)$ and the *calibre* $\mathcal{C}(R)$, allowing us to exhibit an algorithm with a complexity in $O(n^{\mathrm{rk}(R)+\mathcal{C}(R)})$ for searching reactants, while showing that $\mathrm{rk}(R) + \mathcal{C}(R) \leq k$, and that $\mathrm{rk}(R) + \mathcal{C}(R) < k$ most of the time.

The rest of the chapter is organised as follows. Section 2.2 devises a model of a chemical program. In Section 2.3, based on this model, a characterisation of its runtime complexity, by analogy with the subgraph isomorphism problem, is given. Then, we describe the PMJA (Purification of the Minimal Juncture Assignment) algorithm putting this result into practice and discuss its complexity. Finally, Section 2.4 provides a brief discussion about the relevance of *not* reaching inertia, in particular in distributed settings.

2.2 Model

A chemical program can be represented by the triple $CP = (\mathcal{T}, M, \mathcal{R})$, where $\mathcal{T}$ is the set of possible types of molecules, M is the complete multiset of molecules in the initial solution and $\mathcal{R}$ is a set of rules to be applied on M. The general form for $R \in \mathcal{R}$ is:

$$\textbf{replace } x_1 :: T_1, \ldots, x_k :: T_k \textbf{ by } P(x_1, \ldots, x_k) \textbf{ if } C(x_1, \ldots, x_k) \tag{2.1}$$

R is composed of three parts : (1) a *pattern* multiset $x_1 :: T_1, \ldots, x_k :: T_k$ of molecules to find in the solution to apply the rule, where $T_i \in \mathcal{T}$ and x_i is the name of the variable, (2) a multiset of molecules $P(x_1, \ldots, x_k)$ produced by the rule and (3) the reaction's condition $C(x_1, \ldots, x_k)$, a formula of the propositional logic, in which each literal is the application of a boolean function on some of the variables x_1 to x_k. The rule can be applied at run time only if we can find a multiset of molecules satisfying both the pattern and condition. In this chapter, we restrict our study to chemical programs having only one rule. Extending this work to multiple-rules chemical programs does not present major difficulties.

The problem to be solved is to find elements in the multiset satisfying a rule's condition. The algorithm to be designed takes two input parameters, namely, a chemical rule R, as described by Expression 2.1 and a multiset M composed of n molecules. It returns either a tuple $(m_1, \ldots, m_k)$ of molecules in M, where m_i is of type T_i for all i and $C(m_1, \ldots, m_k)$ is true if such a tuple exists in M. Otherwise, it returns $\perp$.

2.2.1 Modeling of the rule's condition

Notice first that the case where the condition is absent is simple to solve, since it is only necessary to compare the number of available molecules for each type to the number of molecules required. Then, recall that, any propositional formula can be put in disjunctive normal form[1], so:

$$C(x_1, \ldots, x_n) = \bigvee_{i=1}^{L} \bigwedge_{j=1}^{l_i} f_{ij}(X_{ij})$$

where, for all i and j, X_{ij} is a subset of variables of R. Since molecules $m_1, \ldots, m_k$ verify $C_1(x_1, \ldots, x_k) \vee C_2(x_1, \ldots, x_k)$ if and only if they verify either $C_1(x_1, \ldots, x_k)$ or $C_2(x_1, \ldots, x_k)$, the various terms of the disjunction can be searched separately. We can consequently replace R by an equivalent set of L rules $\{R_i\}$ where the condition of each rule is one distinct term amongst the L terms of the disjunction, *i.e.,* a conjunction of boolean functions applied to a subset of variables:

$$R_i = \textbf{replace } x_1, \ldots, x_k \textbf{ by } P(x_1, \ldots, x_k) \textbf{ if } f_1(X_1) \wedge \cdots \wedge f_l(X_l). \tag{2.2}$$

While the number L of such rules can be high, it only depends on the condition of the initial rule, and not on the size of the multiset. Also, the inertia is detected if and only if reactants cannot be found for any of the rule. Consequently, the searching of molecules for each rule is independent, and rules can be processed in parallel. Let us introduce some accessors to the elements of a rule R like the one illustrated by Expression 2.2:

- $\mathrm{var}(R) = \{x_1, \ldots, x_k\}$ denotes the set of its variables. $|\mathrm{var}(R)|$ is called the *arity* of R.
- $\mathrm{pred}(R) = \bigcup_{i=1}^{l} \{(f_i, X_i)\}$ denotes the set of predicates to be tested on $\mathrm{var}(R)$. Each predicate $p = (f, X)$, associated with a literal in the condition, has a *function* $\mathrm{func}(p) = f$ and *arguments* $\mathrm{arg}(p) = X \subseteq \mathrm{var}(R)$.

Definition 1 (rank of a rule)**.** *The* rank *of a rule R, denoted by* $\mathrm{rk}(R)$*, is the greatest arity of its predicates:* $\mathrm{rk}(R) = \max_{p \in \mathrm{pred}(R)} |\mathrm{arg}(p)|$.

[1] Note that the type of a molecule can be seen as an individual condition on this molecule.

A rule can be represented by a hypergraph, in which the vertices are the variables and the hyperedges are the predicates, as illustrated in Figure 2.1. Note that most of the problems encountered in the literature on artificial chemistries are solved by rules with a rank of 1, 2 or 3, as their predicates mostly involve comparisons of pairs of variables [39]. When the rank is 2, the representation is a simple graph (as the one on the left in Figure 2.1).

Figure 2.1: The rule **replace** x, y, z, t **by** $P(x, y, z, t)$ **if** $f(z, y) \wedge g(x, z) \wedge h(x, t) \wedge i(y, t) \wedge j(z, t)$ has an arity of 4 and a rank of 2 and can be represented as a graph (left). The rule **replace** x, y, z, t **by** $P(x, y, z, t)$ **if** $f(x, y, z) \wedge g(x, y, t) \wedge h(x, z, t) \wedge i(y, z, t)$ has an arity of 4 and a rank of 3 and needs a hypergraph to be represented (right).

2.2.2 Structuring of the multiset

Until here, we focused on defining the rules. We now devise a model for the multiset of molecules, and how to structure it according to the rule processed. The key concept in the following is the *axiom*. An axiom can be seen as an instantiated predicate. In other words, it is a predicate for which an actual molecule has been found for each of its variables so as to make the predicate true. Note that, as reflected by the chosen term *axiom*, it can be seen as a minimal set of truth in regards to the rule's condition.

Definition 2 (axiom). *Let $p = (f, X)$ be a predicate. An* axiom *is a pair (p, m) where m is a function that associates a molecule to each variable of p, such that $f(m(x_1), \dots, m(x_n))$ is true.*

We extend to axioms, the notations previously defined for predicates. Let an axiom $a = (p, m)$. Then, $\text{func}(a) = \text{func}(p)$ and $\text{arg}(a) = \text{arg}(p)$. Also, $\text{pred}(a) = p$. We define a convenient notation to access to molecules chosen: $a[x] = m(x)$ for all $x \in \text{arg}(p)$ and $\text{mols}(a) = \{a[x] : x \in \text{arg}(p)\}$. Let for instance a predicate $p = (f, X)$ with $X = (x, y)$ and $f(x, y)$ returning true when $x + y = 0$ and false otherwise. Provided they are in the multiset, molecules 3 and -3 can be used to build an axiom $a = (p, m)$, where $m(x) = a[x] = 3$ and $m(y) = a[y] - 3$, and we have $\text{mols}(a) = \{3, -3\}$. We finally extend these notations to sets of axioms: Let A be a set of axioms. The set of molecules of A is the union of molecules in each axiom:

$$\text{mols}(A) = \bigcup_{a \in A} \text{mols}(a)$$

Given a rule R and a set of molecules M, we can define the set of all the axioms that can be constructed.

Definition 3 (set of axioms induced). *For a solution M and a rule R, we define the* set of axioms induced *by R in M as the set of axioms composed of a predicate of R and of molecules of M:*

$$\mathcal{A}(R, M) = \{a : \text{pred}(a) \in \text{pred}(R) \wedge \forall x \in \text{arg}(a), a[x] \in M\} \tag{2.3}$$

Continuing with our example, let us add another predicate p_2 to our rule R such that $p_2(y, z)$ returns true if $z = y^2$, and false otherwise. Assuming $M = \{-3, 3, 6, 9\}$, $\mathcal{A}(R, M)$ will be composed of four axioms a_1, a_2, a_3 and a_4 such that $\text{mols}(a_1) = \{-3, 3\}$, $\text{mols}(a_2) = \{3, -3\}$, $\text{mols}(a_3) = \{-3, 9\}$ and $\text{mols}(a_4) = \{3, 9\}$.

The goal is to associate each variable with a molecule such that this molecule is used in at least one axiom for each predicate of the rule. In this case, the variable is said to be *satisfied*. In our last example, Variable y is satisfied by Molecule 3 (and Molecule -3).

Definition 4 (satisfaction of a variable)**.** *Let $x \in \text{var}(R)$, A be a set of axioms and $m \in \text{mols}(A)$. The molecule m is said to* satisfy *x in A, denoted $m \models_A x$, if for every predicate of the rule pertaining x, we can find at least one axiom in A in which x is associated with m:*

$$\forall p \in \text{pred}(R), x \in \arg(p) \Rightarrow (\exists a \in A, \text{pred}(a) = p \wedge a[x] = m). \tag{2.4}$$

We are interested in finding sets of axioms leading to a possible reaction. A set of axioms specifies a possible reaction if there is a one-to-one relation between its variables and its molecules. Let us characterise sets of axioms so as to be able to define the subsets of axioms that can actually lead to a reaction.

- A set of axioms is *refined* if $\forall m \in \text{mols}(A), |\{x \in \text{var}(R) : m \models_A x\}| \geq 1$
- A set of axioms is *exclusive* if $\forall m \in \text{mols}(A), |\{x \in \text{var}(R) : m \models_A x\}| \leq 1$

Ensuring exclusivity can be done by adding inequality constraints in the rule so the same molecule cannot satisfy several variables. In a both refined and exclusive set of axioms, all molecules are assigned to one and only one variable. This does not mean that it specifies a possible reactions, as some variables may not be satisfied in it. Let us define sets of axioms that can lead to reactions:

- A set of axioms is *reactive* if $\forall x \in \text{var}(R), |\{m \in \text{mols}(A) : m \models_A x\}| \geq 1$
- A set of axioms is *purified* if $\forall x \in \text{var}(R), |\{m \in \text{mols}(A) : m \models_A x\}| \leq 1$

In other words, a *reactive* set of axioms contains at least one subset of axioms that specifies a reaction. Extracting one of these subsets can be done by *purifying* it. The algorithm presented later in Section 2.3 consists in taking the set of induced axioms and trying to assign molecules to variables so as to refine it, and then test the *reactivity* of such an *assignment*:

Definition 5 (assignment)**.** *Let A be a set of axioms, $x_1, \ldots, x_p \in \text{var}(R)$ and $m_1, \ldots, m_p \in \text{mols}(A)$. An* assignment *of $m_1, \ldots, m_p$ to $x_1, \ldots, x_p$, denoted $A' = A[x_1 := m_1, \ldots, x_p := m_p]$, is the largest (in the sense of inclusion) refined subset of A verifying $\forall i \leq p, \forall m, m \models_{A'} x_i \Rightarrow m = m_i$.*

Algorithmically speaking, and as detailed in Section 2.3, given a set A of induced axioms, an assignment of A is obtained by removing all the molecules and axioms from A that cannot be in any refined subset of A, given the set of molecules chosen M_{chosen} for the subset $V_{assigned}$ of variables assigned. Firstly it means, given $V_{assigned}$, removing all the axioms corresponding to predicates containing variables in $V_{assigned}$ but built using molecules not in M_{chosen}. Secondly, it means removing all molecules consequently not used anymore, and the axioms in which they were, making in turn other molecules unused. This refinement process is repeated until no more refinement is needed.

2.2.3 NP-hardness and the subgraph isomorphism problem

The reactants searching problem can be reduced to the subgraph isomorphism problem. Similarly to the hypergraph of the rules, a set of axioms can be modeled by a hypergraph of molecules, where the vertices are the molecules contained in the set and each vertex corresponds to an axiom that links its arguments and is labelled by the predicate it satisfies. Under this formalism, a purified reactive assignment is a sub-hypergraph in the hypergraph of molecules that is isomorphic to the hypergraph of the rule, with respect to the labels of the edges.

The subgraph isomorphism problem is one of the early problems to be known to be NP-complete [33]. This property gives clues on the intrinsic complexity of the reactants searching problem. The subgraph isomorphism problem has been well documented [32] since the first algorithm proposed by Ullman in 1976 [101], which was based on backtracking. In [72], a solution is proposed based on the construction of a decision tree, allowing the search for isomorphic subgraphs in polynomial time, after a pre-treatment in exponential time. This work cannot be applied in our case, as the pre-treatment would have to be made on the graph of the molecules, which constantly changes depending on the reactions arising during the computation.

As shown in [64], it is possible to reformulate the subgraph isomorphism problem as a *Constraint Search Problem* (CSP). The distributed version of CSP, disCSP [23], is very close to the reactants searching problem. The disCSP problem is generally treated through an exhaustive exploration of the nodes of the network [40, 106], possibly with an optimisation using back-tracking [105].

To show that the reactant searching problem is NP-hard, let consider a graph $G = (V, E)$. Let us define a chemical program with $M = V$ and the following rule:

$$R = \textbf{replace } x_1, \ldots, x_k \textbf{ by } P(x_1, \ldots, x_k) \textbf{ if } \bigwedge_{i=1}^{k-1} \bigwedge_{j=i+1}^{k} (x_i, x_j) \in E$$

This rule's condition is a conjunction of predicates on every possible pairs of the rule's variables. Consequently, it can be applied on M if and only if G contains a clique. This shows that the reactants searching problem is NP-hard, depending on its arity k. It is actually NP-complete under the assumption that the evaluation of reaction conditions terminates in a time which is independent of both n and k. If this is the case, a non-deterministic Turing machine can do all tests between molecules in polynomial time.

The rule used to show the NP-hardness of the reactants searching problem has a rank of 2, and its graph is a clique. In other words, it can be considered as a complicated rule since its reaction condition has as many literals as there are pairs of variables. We should therefore find a way to characterize the complexity of a rule. This is what we do in the next section, which provides a study of the *calibre* of a rule, and an algorithm solving the reactants searching problem, based on it.

2.3 Calibre and the PMJA Algorithm

In this section, we present an algorithm for the reactants searching problem, based on a characterization of the complexity of a rule, using the notion of *calibre* of the rule. Then, we present the PMJA algorithm, which levers this characterisation to allow for a better complexity than the basic $O(n^k)$ case, most of the time. For brevity, we will not exhibit the complete proofs of properties and theorems in this document. Please refer to the research report [18] for the details.

2.3.1 Calibre of a rule

Determining the calibre of a rule relies on determining its minimal *juncture*:

Definition 6 (juncture of a rule)**.** *Assuming the variables of a rule are completely sorted by an order $\lessdot$, we define the* juncture *of R for the order $\lessdot$, denoted $\mathcal{J}_{\lessdot}(R)$, as the set of variables which are not the smallest in several of their predicates:*

$$\mathcal{J}_{\lessdot}(R) = \{x \in \mathrm{var}(R) : |\{p \in \mathrm{pred}(R) : \exists y \lessdot x, \{x, y\} \subset \arg(p)\}| \geq 2\}\,. \tag{2.5}$$

Definition 7 (calibre of a rule)**.** *The* calibre *of a rule is the size of its smallest juncture considering all possible orders:*

$$\mathcal{C}(R) = \min_{\lessdot} |\mathcal{J}_{\lessdot}(R)|. \tag{2.6}$$

A juncture $\mathcal{J}_{\lessdot}(R)$ such that $|\mathcal{J}_{\lessdot}(R)| = \mathcal{C}(R)$ is said to be minimal.

Let us illustrate the two previous definitions. As rules with a rank of 2 represent most of rules found in chemical programs in practice, we will discuss the calibre of some of these rules, by having a look at their corresponding graph:

- The calibre of a rule having a tree shape is 0. By definition, each node of a tree has a single parent except the root which is an orphan. Therefore, following the topological order, we find no node in the potential juncture.

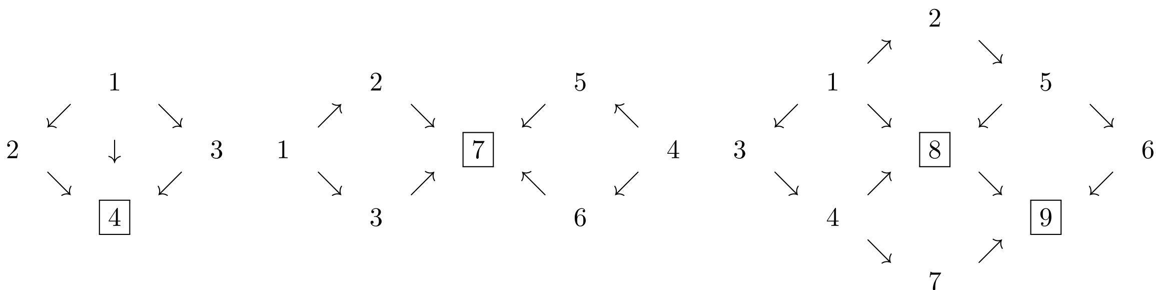


Figure 2.2: The *bridge*, the *eight* and the *lattice* with their minimal juncture.

- The calibre of a rule whose graph is a cycle shape is one. On one hand, regardless of the order chosen, the greatest element has necessarily two smaller neighbours: its predecessor and its successor in the cycle, making the calibre is at least equal to 1. On the other hand, for an order that follows the cycle, all the other elements have 0 or 1 parent, so the calibre is at most 1.
- As illustrated in Figure 2.2, other examples of graphs include the *bridge* and the *eight*, whose calibre is 1, as well as the *lattice*, whose calibre is 2.

Let us compare the calibre of a rule to its arity. As detailed soon, the complexity of our algorithm depends on $\mathcal{C}(R) + \mathrm{rk}(R)$. It is possible to group predicates: it is equivalent to search for reactants x and y that verify both $f(x, y)$ and $g(x, y)$ or only $(f \wedge g)(x, y)$. This grouping may have an effect on both $\mathcal{C}(R)$ and $\mathrm{rk}(R)$. A direct remark is that, if we group all the predicates into a single one, we have $\mathrm{rk}(R) = |\mathrm{var}(R)|$ and $\mathcal{C}(R) = 0$, so it is always possible to find a grouping such that $\mathcal{C}(R) + \mathrm{rk}(R) \leq |\mathrm{var}(R)|$. The particular case of rules of rank 2 requires no particular grouping:

Theorem 1 (upper bound on the calibre of a rule with a rank of 2)**.** *Let R be a rule of rank 2. The following inequality is verified, with equality if and only if the graph of R is a clique:*

$$\mathcal{C}(R) + 2 \leq |\mathrm{var}(R)|.$$

Proof. Regardless the shape of the graph and the order chosen, the two smallest variables have at most one smaller neighbour, so they cannot be part of any juncture. Let us be more precise: either the graph is a clique or it is not. If the graph is a clique, then all other variables have at least these two nodes as smaller neighbours, and so are in the juncture, whatever the order is chosen, in which case the calibre is exactly equal to the number of variables minus 2.

Conversely, if the graph of R is not a clique, then there exists x and y that are not connected. Let R' be a rule with the same variables as R and a predicate connecting all pairs of variables, except (x, y). Firstly, let us remark that $\mathcal{C}(R) \leq \mathcal{C}(R')$. Secondly, let z be a variable different from x and y, and $\lessdot$ an order in which the three smallest variables are $z \lessdot x \lessdot y$ in that order. Then, $\mathcal{C}(R') = |\mathrm{var}(R)| - 3$, so $\mathcal{C}(R) \leq \mathcal{C}(R') \leq k - 3$. □

Theorem 2 (purification of the juncture's assignment)**.** *Let R be a rule of arity k whose variables are ordered by $\lessdot$ with $\{x_1, \ldots, x_c\} = \mathcal{J}_{\lessdot}(R)$. Let A be a set of axioms. For all $m_1, \ldots, m_c \in \mathrm{mols}(A)$, $A_c = A[x_1 := m_1, \ldots, x_c := m_c]$ is reactive if and only if there are $m_{c+1}, \ldots, m_k \in \mathrm{mols}(A)$ such that $A_k = A[x_1 := m_1, \ldots, x_n := m_k]$ is reactive and purified.*

Proof. If A_k is reactive, given that $A_k \subset A_c$, clearly A_c is reactive too. Conversely, suppose A_c is reactive. The goal is to show that A_c can get purified by choosing one and only one molecule for every variable. This can be done by choosing one molecule for each variable, one by one, following the order $\lessdot$. When choosing a molecule for x_i, three cases can occur, depending on the number of predicates containing x_i and a smaller variable x_j:

1. If x_i has no smaller neighbour, since A_c is reactive, there exists at least one molecule $m_i \models_{A_c} x_i$. We can choose any one of them.

2. If there is one (and only one) such predicate p, since A_c is reactive, there is at least one axiom a such that $\text{pred}(a) = p$ in A_c. Then, one can choose $m_i = a[x_i]$.

3. Otherwise, $x_i \in \mathcal{J}_\lessdot(R)$, so x_i is already assigned in A_c. Since the assignment is reactive, there exists $m_i \in \text{mols}(A_c)$ that is suitable with all the already chosen variables.

□

2.3.2 The PMJA algorithm

In this section, we present the PMJA (Purification of the Minimal Juncture Assignment) algorithm that solves the reactants searching problem. Algorithm 1 shows the global PMJA algorithm for a rule R, with an arity k and a juncture $\mathcal{J}_\lessdot(R) = \{x_1, \ldots, x_c\}$. It takes as argument a set of axioms A of type **AxiomSet** organised like a graph when the rank is 2. A gives access to all the molecules used in these axioms, sorted by the variables they satisfy. Each molecule gives in turn access to a set of references to the axioms in which it is an argument, sorted by predicate. In terms of implementation, an AxiomSet could be implemented through a structure having: 1) a hash table of molecules, where a molecule is retrieved using the variable it satisfies as the key, 2) a hash table of the axioms these molecules satisfy retrieved using the predicate they implement as a key, 3) cross-references from molecules to axioms, and from variables to predicates.

According to this structuring, the physical size of the AxiomSet A for a rule R and a set of molecules M, is in $O(|\mathcal{A}(R, M)| + |\text{mols}(\mathcal{A}(R, M))|)$, and getting predicates and molecules from variables as well as getting axioms from molecules can be assumed to be done in constant time, apart from cloning the structure itself which is necessarily linear in the size of A. By convention, the indices of an array vary between 1 and tab.size.

Algorithm 1: Reactants searching for R with $\text{var}(R) = \{x_1, \ldots, x_k\}$ and $\mathcal{J}_\lessdot(R) = \{x_1, \ldots, x_c\}$.

```
1 Molecule[] : findReactants(AxiomSet A) :
2     forall the m_1 ⊨ x_1, ..., m_c ⊨ x_c do
          // m_i variables are global
3         AxiomSet A' ← A.clone()
4         buildAssignment(A')
5         if ¬(refineAndTestReactivity(A')) then
6             continue
7         return(purify(A'))
```

Algorithm 2: Assignment building given an AxiomSet.

```
1 AxiomSet : buildAssignment(AxiomSet A) :
      // A is passed by reference
2     for i ← 1 to c do
3         forall the Molecule m ∈ A.molecules(i) s.t. m ≠ m_i do
4             m.removed ← true
5             forall the Axiom a ∈ m.axioms(*) do
6                 a.removed ← true
```

Algorithm 1 is based on Theorem 2, that suggests to test the reactivity of all the possible assignments of a juncture to detect inertia. The details of the functions it calls are detailed in Algorithms 2, 3 and 4. It is composed of a main loop, which is executed once for each tuple of molecules $(m_1, \ldots, m_c)$ that can potentially be an assignment of $\mathcal{J}_\lessdot(R)$. More precisely, as can be seen in Algorithm 1, the loop is composed of two main parts:

Algorithm 3: Refining and testing the reactivity of an AxiomSet.

```
1  Boolean : refineAndTestReactivity( AxiomSet A) :
      // A is passed by reference
2     Boolean reactive, changed
3     repeat
4        changed ← false
5        for i ← 1 to k do
6           reactive ← false
7           forall the Molecule m ∈ A.molecules(i) s.t. ¬m.removed do
8              reactive ← true
9              forall the Predicate p ∈ x_i.predicates() do
10                Axiom at[ ] ← m.axioms(p)
11                while m.first(p) ≤ at.size ∧ at[m.first(p)].removed do
12                   m.incrementFirst(p)
13                if m.first(p) > at.size then                    // m.axioms(p) is empty
14                   changed ← true
15                   m.removed ← true
16                   forall the Axiom a ∈ m.axioms(*) do
17                      a.removed ← true
18          if ¬reactive break
19    until ¬reactive ∨ ¬changed
20    return(reactive)
```

Algorithm 4: Purifying a reactive AxiomSet.

```
1  Molecule [] : purify(AxiomSet A) :
2     Molecule M[k]
3     for i ← 1 to k do // x_σ(1) ⋖ ··· ⋖ x_σ(k)
4        switch {p ∈ pred(R) : ∃x_σ(j) ⋖ x_σ(i), {x_σ(i), x_σ(j)} ⊂ arg(p)} do
5           case ∅ : // choose any
6              forall the Molecule m ∈ A.molecules(σ(i)) do
7                 if ¬m.removed then  M[i] ← m; break;
8              break
9           case {p} : // x_σ(i) ⋗ x_σ(j) ∈ arg(p)
10             M[i] ← M[j].axioms(p)[M[j].first(p)].get(x_i); break
11          otherwise // x_σ(i) ∈ J_⋖(R)
12             M[i] ← m_i;
13    return M
```

1. In the first part of the loop (Lines 3-6), the assignment $A' = A[x_1 \leftarrow m_1, \ldots, x_c \leftarrow m_c]$ is computed. If it is not reactive, it is dropped. The assignment is computed by a successive elimination of molecules, through two steps:

   a) First, through the use of the *buildAssignment()* function, the assignment is built from the AxiomSet. As detailed in Algorithm 2, this function takes the cloned AxiomSet by reference, and removes all molecules that were not chosen from it. Subsequently, all axioms the removed molecule were in are removed on their turn.

   b) Secondly, through the use of the *refineAndTestReactivity()* function detailed in Algorithm 3, the built assignment, still stored in the same AxiomSet variable is refined by removing all the molecules that cannot be in any refined subset of A'. It is done by repeatedly removing molecules that are not a member of any axiom left after removals made in the *buildAssignment()* function. When molecules are removed, all the axioms they took part in are also removed, making it potentially possible to remove other molecules, and so on, until either we cannot find any more non-used

molecule or some variable cannot get satisfied anymore.[2] Remind that the axiomSet passed as a parameter to both *buildAssignment()* and *refineAndTestReactivity()* is passed by reference. In addition to refining the AxiomSet, the *refineAndTestReactivity()* function finally returns *true* if the refined AxiomSet is reactive, *false* otherwise.

2. The second part in Algorithm 1 starts after the refinement in case the *refineAndTestReactivity()* function returned *true*. Indeed, if all molecules are still satisfied, according to Theorem 2, this assignment of the juncture can be purified so as to make a reaction. The *purify()* function, detailed in Algorithm 4, achieves this purification and returns a set of molecules that can react. This set is finally returned by the global algorithm.

If no assignment of the juncture is reactive, the solution is inert and the algorithm returns $\perp$. From these observations, it can be inferred that the algorithm correctly returns a reactive purified assignment if there is one, and $\perp$ otherwise.

Proposition 1 (Time complexity)**.** *In the worst case, the time complexity of Algorithm 1 is:*

$$T(R, A) = \mathcal{O}\left(|\text{mols}(A)|^{\mathcal{C}(R)+\text{rk}(R)}\right).$$

Proof. If the solution is inert, which is the worst case for searching reactants, there are $\prod_{x \in \mathcal{J}_{\lessdot}(R)} |\{m \models x\}| \leq |\text{mols}(A)|^{\mathcal{C}(R)}$ executions of the main loop. We will now show that the complexity of one iteration of the main loop is proportional to the size of the axiom set.

- In the *buildAssignment()* function, where the molecules that do not match the current choice are removed, each axiom is considered at most once for each argument, leading to a complexity in $\text{rk}(R) \times |A|$, where $|A|$ is to be interpreted as the number of axioms, not the size of the AxiomSet structure.
- The complexity of the *refineAndTestReactivity()* function is defined by the complexity of its two inner loops. On Line 11 of Algorithm 3, the field $m.first(p)$ can only grow, so each axiom is again checked only once for each molecule in its arguments. In the loop of Line 16, an axiom can only be removed once for each argument.
- The complexity of the *purify()* function is only $|\text{var}(R)| \ll |A|$.

To sum up, the complexity of the main loop does not exceed $\text{rk}(R) \times |A|$ and is executed less than $|\text{mols}(A)|$ times, so the complexity of the whole algorithm is at most $|\text{mols}(A)|^{\mathcal{C}(R)} \times \text{rk}(R) \times |A|$. Given that:

$$|A| \leq \binom{|\text{mols}(A)|}{\text{rk}(R)} \leq \frac{|\text{mols}(A)|^{\text{rk}(R)}}{\text{rk}(R)!},$$

we can finally bound the complexity by

$$\begin{aligned} T(R, A) &= \frac{|\text{mols}(A)|^{\text{rk}(R)}}{\text{rk}(R)!} \times |\text{mols}(A)|^{\mathcal{C}(R)} \times \text{rk}(R) \\ &= \frac{1}{(\text{rk}(R)-1)!} |\text{mols}(A)|^{\mathcal{C}(R)+\text{rk}(R)} \\ &= \mathcal{O}\left(|\text{mols}(A)|^{\mathcal{C}(R)+\text{rk}(R)}\right). \end{aligned}$$

□

[2] For each molecule m and predicate p for which m can be an argument, we keep an index on the first non-removed axiom corresponding for p and containing m, with $m.\text{first}(p)$ initialised to 1, in order to check efficiently if a molecule must be removed.

We have seen that it was possible to modify any rule R so that $\mathcal{C}(R)+\text{rk}(R) \leq |\text{var}(R)|$. This establishes that the proposed algorithm has a complexity which is either similar or better than $\mathcal{O}(n^k)$. Note that in practice, we have actually $\mathcal{C}(R)+\text{rk}(R) < |\text{var}(R)|$ most of the time. Note also that this gain can be evaluated at compile time (it depends on the rank and the calibre of a rule, both obtained by static analysis of the rule).

2.4 Discussion

2.4.1 Summary

Chapters 1 and 2 presented and illustrated the chemical programming style and the HOCL programming language. Secondly, it focused on multiset rewriting, the main concept used to enact the paradigm behind the chemical metaphor. We have shown, through a static analysis of the reaction condition, that multiset rewriting, using a rule of arity k can be solved in a time bounded by $n^{\text{rk}(R)+\mathcal{C}(R)}$, with $\text{rk}(R) + \mathcal{C}(R) \leq k$, and we exhibited the PMJA algorithm for this. For rules of rank 2, we were able to characterise the case of equality. An interesting point is that we are able to estimate the execution complexity at compile time, which makes it possible to provide the programmer with optimisation recommendations.

This work opens questions. In particular, there is a need to deal with compilation. So far we have only presented the effective search of molecules when the graph of the rule is known. We used arguments from logics to claim that compilation was indeed possible. It would be interesting to find efficient algorithms to choose an optimal order of the variables in the rule. Choosing an order for which the juncture is minimal needs to be done only once, at compile time, while providing guarantees on the complexity of the algorithm. Note that, however, the algorithm is flexible and works regardless of the juncture used. The complexity could be further reduced by a finer choice of the order, with respect to the quantity of molecules for each variable. (Among several minimal junctures, prefer those for which the number of actual molecules is minimised.) Note also that this work did not deal with higher-order. This subject will be tackled later in Chapter 4, but in distributed settings.

2.4.2 Approximation and distribution

The above work is fully relevant only if you actually need to reach inertia: the PMJA algorithm is a deterministic inertia checker: if it returns $\perp$, it means that it is no longer possible to find reactants in the solution. Depending on what is to be computed, reaching inertia may not be worth its cost. Sometimes, a sufficiently precise estimation of the result can be obtained without having to reach inertia. Ergo, relaxing this inertia constraint allows to define lazier approaches to make molecules react, the key point being that globally, we *do our best* to find reactants. At some point in the execution, the engine can decide to stop searching for reactants without being sure that some reactants cannot be found anymore.

In distributed settings, gossip protocols [61] perfectly illustrate this. They are simple local information exchange protocols, that allow to compute an aggregated global value from dispersed small bits of information. In a gossip protocol, every node periodically exchanges information with another node uniformly chosen at random among the set of its neighbours. Provided the neighbours are well chosen, and the periodicity of exchanges is relatively uniform among nodes, a gossip protocol can converge exponentially quickly, independently of the network size, towards the actual aggregated value [59].

A simple example of a gossip protocol is the estimation of the size of the network: every node starts with value 0 except one which starts with value 1. One round consists in exchanging values (with one neighbour) and then locally computing an estimated average value. Quickly, the value hold by one node will tend towards some $v \approx 1/N$. The last step consists in everyone calculating $1/v$ to find a local approximation of N. Such a simple calculation is easily translatable into a chemical program:

$$\textbf{let } \textit{count} = \textbf{ replace } x, y \textbf{ by } (x+y)/2, (x+y)/2$$

Applying this rule repeatedly on a set of values which is initially a multiset of many 0s and a single 1 will result in having many good approximations of $1/N$, N being the cardinality of the multiset. In this example,

inertia cannot be reached. We can always find two molecules on which to apply the rule (even if a reaction does not modify the values of molecules chosen).

The previously described gossip protocol can be seen as a distributed implementation of this chemical program. Note that the atomic capture is easily ensured as a node cannot communicate with multiple nodes at the same time. It can start a new gossip round only after the previous one is completed. Gossiping is actually a very good candidate when it comes to build a *distributed chemical machine*. The key point in such a venture is to build a *stirrer* to disseminate molecules across nodes so that potential reactants can meet on a node and become actual reactants. Of course, the highest the arity of a rule is, the more difficult it is to have all the needed reactants together on one node, and more complex gossiping strategies need to be elaborated. The challenges of a gossip-based stirrer was explored in a work in collaboration with Antoine Chatalic, whose results are available in [30].

In the two following chapters, we devise the construction of a distributed chemical machine, based on a distributed hash table for molecule discovery, and on a custom algorithm for a distributed adaptive atomic capture of molecules. The runtime environment to be presented does not take the bias of approximation and attempts at providing the capability of inertia detection.

Chapter 3

Adaptive Atomic Capture of Multiple Molecules

In this chapter, we address the problem of atomic capture in distributed settings. More concretely, it means that a set of nodes can apply the rules in parallel on the mutiset, itself distributed. It constitutes an important milestone towards the construction of a distributed chemical machine. We devise an adaptive capture protocol that takes into account the changing density of reactants over the course of the execution. When the density is high, an optimistic, light protocol is used. When it drops, a pessimistic approach ensures the liveness of the system. Protocol's properties are formally established and its behaviour is studied through an extensive set of simulations.

Joint work with:

- ⋆ Marko Obrovac (PhD student, Inria)
- ⋆ Marin Bertier (Associate professor, INSA de Rennes)

3.1 The importance of being atomic

The *max* example given in Chapter 2 does not need the atomic capture to output a correct result. If an integer takes part in several reactions at the same time, due to the fact that the highest value will not disappear anyway in a reaction, the execution will actually get speeded up without any problem. This is not true for other kinds of aggregation such as *counting* aggregations. Let us give another example of a chemical program mimicking a MapReduce program that counts the number of characters in a text composed of a set of words:

$$\begin{array}{l} \textbf{let } count = \textbf{ replace } s :: string \textbf{ by } len(s) \textbf{ in} \\ \textbf{let } aggregate = \textbf{ replace } x :: int, y :: int \textbf{ by } x + y \textbf{ in} \\ \quad \langle "maecenas", "ligula", "massa", "varius", "a", \\ \quad "semper", "congue", "euismod", "non", "mi", \\ \quad count, aggregate\rangle \end{array}$$

The rule named *count* consumes a *string* element and introduces an integer representing its length into the solution. The *aggregate* rule consumes two integers to produce their sum. By its repeated execution, this rule aggregates sums to produce the final count. At run time, these rules are executed repeatedly and concurrently, the first one producing inputs for the second one. Again, while the result of the computation is deterministic, the order of its execution is not. A possible execution is the following. Let us consider that

the first rule is applied on the first three strings as represented above, and on the last one. The state of the multiset is then the following (rules are omitted for the sake of brevity):

$$\langle\ "varius", "a", "semper", "congue", "euismod", "non", 8, 6, 5, 2\ \rangle.$$

Then, let us assume, still arbitrarily, that the *aggregate* rule is triggered three times on the previously introduced integers, producing their sum. Meanwhile, the remaining strings are scanned by the *count* rule. The multiset is then:

$$\langle\ 6, 1, 6, 6, 7, 3, 21\ \rangle.$$

With the repeated application of the *aggregate* rule, inertia is reached and the final solution is:

$$\langle\ 50\ \rangle.$$

In this example, the atomic capture is clearly a fundamental condition. If the same string were to be captured by different nodes running the *count* rule in parallel, the count for a word would appear more than once in the solution, leading to an incorrect result.

Let us slightly refine the problem to be tackled in this chapter: we consider a chemical program made of a multiset of objects (molecules), and a set of rules to be applied concurrently on them. Both the objects and the rules are distributed over a set of nodes on which the program runs. Each node periodically tries to fetch molecules needed for the reactions it is trying to perform. As several molecules can satisfy the pattern and conditions of several reactions performed concurrently by different nodes, the same molecule can get potentially requested by several nodes at the same time, inevitably leading to conflicts. Mutual exclusion on the molecules is thus mandatory. Our problem resembles some classic mutual exclusion problems, in particular the drinking philosopher's problem [29] and *k-out of-M* problem [85]. Still, it differs from them in several aspects. Firstly, the molecules are interchangeable to some extent. The requested molecules must match a pattern defined in the reaction rule a node wants to perform; if two molecules, say A and B, both match the rule's pattern, any of the two may be used in the actual reaction. Then, we differentiate two processes which are:

1. finding molecules matching a pattern (achieved by a *discovery protocol*);

2. obtaining/allocating them to perform reactions (achieved by a *capture protocol*).

Consequently, if one node cannot manage to grab some specific molecules, it will switch to another set of molecules. We are not so much interested in avoiding one node's starvation as in the liveness of the whole system : *some* node should be able to perform one reaction in a finite time in order to move the computation forward.

Secondly, the resources (the molecules) are dynamic : molecules are deleted when they react, and new ones are created. Likewise, the number of resources/molecules (and of possible reactions) will fluctuate over time. This fluctuation can significantly influence the efficiency of the capture protocol. Bear in mind that once the holder of a matching molecule is located, the scale of the network is of less importance, since only the nodes requesting the molecules and their holders are involved in the capture protocol.

To sum up, our objective is to define a protocol for the atomic capture of multiple molecules that dynamically and efficiently adapts to the density of reactants in the system.

Contribution. The protocol proposed in this chapter combines two sub-protocols inspired by previous works on distributed resource allocation, and adapted to the distributed runtime of *chemical* programs. The first sub-protocol, called the *optimistic* protocol, assumes that the number of molecules satisfying some reaction's pattern and condition is high, so only few conflicts for molecules will arise, nodes being likely to be able to grab distinct sets of molecules. While this protocol is simple, fast, and has a limited communication overhead, it does not ensure liveness when the number of conflicts increases. The second one, referred to as *pessimistic* is slower, more costly in terms of communication, but ensures liveness in the presence of an arbitrary number of conflicts (nodes trying to capture sets of molecules with an overlap). Switching from one

protocol to the other is achieved in a scalable, distributed fashion and is based on local success histories in grabbing molecules. Furthermore, we analyse chemical programs containing multiple rules and the possible input/output dependencies they might have and propose a rule-changing mechanism instructing nodes as to which rule to execute. A proof of the protocol's correctness is given, and its efficiency is discussed through a set of simulation results.

Outline. The next section presents the system model used throughout the chapter. Section 3.3 details the sub-protocols, their coexistence, and the mechanism allowing to switch between them. It also proposes a communication-minimisation scheme when the number of conflicts is high as well as the rule-selection mechanism. Proofs of safety and liveness are also given. Then, Section 3.4 presents the simulation results and discusses the efficiency and overhead of the protocol.

3.2 System model

We consider a distributed system $\mathbb{DS}$ consisting of n nodes communicating via message passing. They are interconnected in such a way that a message sent from a node can be delivered, in finite time, to any other node in $\mathbb{DS}$. Moreover, we suppose that a communication channel between any two nodes is a FIFO queue — a message sent at time t is always delivered strictly before a message sent at time $t + \epsilon$. At large scale, such a fully-connected network can be built by relying on a P2P susbtrate such as an overlay network protocol as used for distributed hash tables (DHTs) [88, 96]. They allow us to focus on the atomic capture of molecules and to not worry about the underlying communications' details.

3.2.1 Data and Rules Dissemination

In the following, we assume data and rules are distributed over the nodes. Even if data and rules are initially held by a single external application node, it can contact any of the nodes taking part in the environment and transfer it the chemical solution to be executed. This node in turn can start spreading the molecules and rules to the other nodes. For the rules, a simple out-of-band mechanism, such as a web server from which to download them, can be used. Several nodes will try to apply, at the same time, a subset of the rules of the program (each rule being present in the subset of at least one node). In contrast to rules dissemination, we assume only one node is responsible to grant access to a particular data item (would it exist one or several copies of the item). A DHT can then typically achieve this distribution, each data item being hashed and inserted into the DHT. Chapter 4 focuses on the actual implementation of a distributed chemical machine, and discusses the distribution of rules and data in more details. In the rest of the chapter, we simply assume that every rule of the program is present on all of the nodes in the system, and that each data item is managed by one specific *holder*.

3.2.2 Discovery Protocol

In order for the reaction to happen, a suitable combination of molecules has to be found. While the details of the discovery process are also abstracted out in the remainder of this chapter, they deserve to be preliminarily discussed. The basic *lookup* mechanism offered by DHTs allows the retrieval of an object according to its (unique) identifier. Unlike the *exact match* functionality provided by DHTs, we require nodes to be able to find *some* molecule satisfying a pattern (*e.g.*, one *integer*) and condition (*e.g.*, *greater than* 3). This can be done through range queries on top of the overlay network, *i.e.,* mechanisms to find some molecule(s) falling within a range, provided the molecules are ordered on a (possibly complex, multi-dimensional) criterion, as for instance provided in [89]. This mechanism can be easily extended to support patterns and conditions involving several molecules. For instance, when trying to capture two molecules, we need to construct the range query to be sent over the DHT based on the given rule and a firstly obtained molecule. If a second molecule is found matching the rule when associated to the first molecule, the capture protocol is triggered.

3.2.3 Fault tolerance

DHT systems inherently provide fault-tolerance mechanisms. If nodes join, leave or crash, the properties of the communication pattern will be preserved. On top of that, we assume that there exists a higher-level resilience mechanism which prevents the loss of molecules, such as state machine replication [70, 90]. Each node replicates its complete state — the molecules and its current actions — across k neighbouring nodes, where the definition of a *neighbour* depends on the actual DHT scheme used. Thus, in case of a node's failure, one of its neighbours is able to detect it and assume its responsibilities to continue the computation.

3.3 Protocol

Here, the protocol in charge of the atomic capture of molecules is discussed. The protocol can run in two modes, based on two different sub-protocols: an *optimistic* one and a *pessimistic* one. The former is a simplified sub-protocol which is employed while the number of possible reactions is high enough to render the possibility of conflicts negligible. When the reaction rate drops below a given threshold, the pessimistic sub-protocol is activated. While being the heavier of the two in terms of network traffic, this sub-protocol ensures the liveness of the system, even when an elevated number of nodes compete for the same subset of molecules.

3.3.1 Pessimistic sub-protocol

To some extent inspired by the three-phase commit protocol [93] used in database systems, this sub-protocol ensures that at least one node wanting to execute a reaction will succeed. Molecule fetching is done in three phases — the *query*, *commitment*, and *fetch* phases — and involves at least two nodes — the node requesting the molecules, called *requester*, and at least one node holding the molecules, called *holder(s)*. Figure 3.1 delivers the time diagram of a pessimistic molecule fetching (where the requester depicted is the one which is granted the molecule). Algorithms 5 and 6 gives the pseudo-code of each side of the protocol, respectively. Note that a node acts at times as a requester (when it executes rules), and at others as a holder (when it holds a molecule requested by another node).

When molecules suitable for a reaction have been found using the discovery protocol (Line 1 in Algorithm 5), the query phase begins (Line 10). The requester sends *QUERY* messages asynchronously to all of the holders to inform them it is interested in their molecule. Depending on the state of the molecule requested, each of the holders evaluates separately the received message (Lines 1–13 in Algorithm 6) and replies with one of the following messages:

- *RESP_OK*: the requested molecule is available;
- *RESP_REMOVED*: the requested molecule no longer exists;
- *RESP_TAKEN*: the molecule has already been promised to another node.

Unless it received only *RESP_OK* messages, the requester aborts the fetch and sends *GIVE_UP* messages to holders, informing them it no longer intends to fetch their molecules (Line 14 in Algorithm 5).

Following the query phase is the commitment phase, when the requester tries to secure its position by asking the guarantee from the holders that it will be able to fetch the molecules (line 19 in Algorithm 5). It does so using *COMMITMENT* messages. Upon its receipt, each holder sorts all of the requests received during the query phase (line 14 in Algorithm 6) according to the conflict resolution policy (described below). Holders reply, once again, with *RESP_OK*, *RESP_REMOVED* or *RESP_TAKEN* messages. A *RESP_OK* response represents a holder's commitment to deliver its molecule in the last phase. Thus, subsequent *QUERY* and *COMMITMENT* requests from other nodes will be resolved with a *RESP_TAKEN* message. Naturally, if a requester does not receive only *RESP_OK* responses to its *COMMITMENT* requests, it aborts the fetch with *GIVE_UP* messages. The holder then removes the requester from the list, in this way allowing others to fetch the molecule.

Finally, in the fetch phase, the requester issues *FETCH* messages, upon which holders transmit it the requested molecules using *RESP_MOLECULE* messages. From this point on, holders issue *RESP_REMOVED* messages to nodes requesting the molecule.

Pessimistic conflict resolution. Each of the holders individually decides to which requester a molecule will be given. Since we want at least one requester to be able to complete its combination of molecules, all holders apply the same conflict resolution scheme, based on the total order of requesters (Lines 20-27 in Algorithm 6). Any total order scheme can be applied. We here briefly detail a dynamic scheme based on load-balancing: each of the messages sent by requesters contains two fields — the requester's identifier and the number of reactions it has completed so far. When two or more requesters are competing for the same molecule, holders give priority to the requester with the lowest number of reactions. In case of a dispute, the requester with a lower node identifier (ensured to be unique by the DHT's hash function) gets the molecule. Such a conflict resolution scheme promotes fairness while at the same time balancing the workload amongst nodes, seeing that the less reactions a node has done the greater the chances are for it to capture the molecules it needs for a reaction.

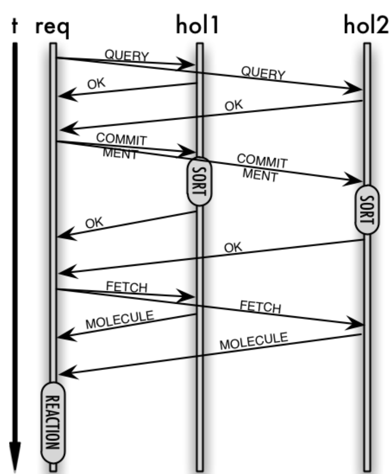


Figure 3.1: Pessimistic exchanges.

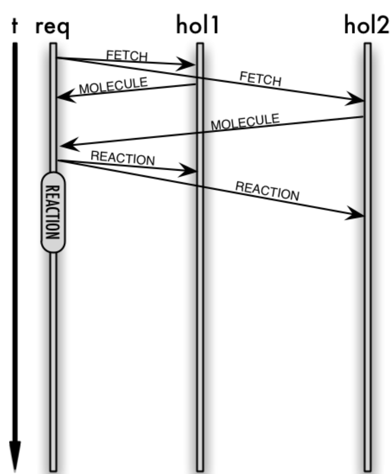


Figure 3.2: Optimistic exchanges.

3.3.2 Optimistic sub-protocol

When the probability of successful multiple concurrent reactions is high (in the sense that they involve distinct sets of molecules), the atomic fetch procedure can be relaxed and simplified by adopting a more optimistic approach. The optimistic sub-protocol requires only two phases — the *fetch* and the *notification* phases. The time diagram of the process of obtaining molecules (when successful) is depicted in Figure 3.2. Algorithm 7 describes the sub-protocol on the requesters' side, while Algorithm 8 describes it on the holders' side.

Once a node acquires information about suitable candidates, it immediately starts the fetch phase (Line 1 in Algorithm 7). It dispatches *FETCH* messages to the appropriate holders. As with the pessimistic sub-protocol, the holder can respond using one of the three previously described types of messages (*RESP_MOLECULE*, *RESP_TAKEN* and *RESP_REMOVED*) as shown in Algorithm 8. A holder that replied with a *RESP_MOLECULE* message, replies with *RESP_TAKEN* messages to subsequent requests until the requester either returns the molecule or notifies it a reaction took place.

Algorithm 5: Pessimistic sub-protocol: requester.

```
1  on event combination found
2      QueryPhase(combination)
3  on event response received
4      if phase = query then
5          QueryPhaseResp(resp_mol)
6      else if phase = commitment then
7          CommitmentPhaseResp(resp_mol)
8      else if phase = fetch then
9          FetchPhaseResp(resp_mol)
10 begin QueryPhase(combination)
11     phase ⇐ query
12     foreach molecule in combination do
13         dispatch QUERY(molecule)
14 begin QueryPhaseResp(resp_mol)
15     if resp_mol ≠ RESP_OK then
16         Abandon(combination)
17     else if all responses have arrived then
18         CommitmentPhase(combination)
19 begin CommitmentPhase(combination)
20     phase ⇐ commitment
21     foreach molecule in combination do
22         dispatch COMMITMENT(molecule)
23 begin CommitmentPhaseResp(resp_mol)
24     if resp_mol ≠ RESP_OK then
25         Abandon(combination)
26     else if all responses have arrived then
27         FetchPhase(combination)
28 begin FetchPhase(combination)
29     phase ⇐ fetch
30     foreach molecule in combination do
31         dispatch FETCH(molecule)
32 begin FetchPhaseResp(resp_mol)
33     add resp_mol to reaction_args
34     if all responses have arrived then
35         Reaction(reaction_args)
36 begin Abandon(combination)
37     phase ⇐ none
38     foreach molecule in combination do
39         dispatch GIVE_UP(molecule)
```

Algorithm 6: Pessimistic sub-protocol: holder.

```
1  on event message received
2      if message = GIVE_UP then
3          remove sender from molecule.list
4      else if message.molecule does not exist then
5          reply with RESP_REMOVED
6      else if message = FETCH then
7          clear molecule.list
8          reply with molecule
9      else if molecule has a commitment then
10         reply with RESP_TAKEN
11     else if message = QUERY then
12         add sender to molecule.list
13         reply with RESP_OK
14     else if message = COMMITMENT then
15         SortRequesters(molecule)
16         if molecule.locker = sender then
17             reply with RESP_OK
18         else
19             reply with RESP_TAKEN
20 begin SortRequesters(molecule)
21     foreach pair of requesters in molecule.list do
22         if req_j.no_r < req_i.no_r then
23             put req_j before req_i
24             continue
25         if req_j.id < req_i.id then
26             put req_j before req_i
27     molecule.locker ⇐ molecule.list (0)
```

If the requester acquires all of the molecules, the reaction is subsequently performed, and the requester sends out *REACTION* messages to holders to notify them the molecules are being consumed. This causes holders to reply with *RESP_REMOVED* messages to subsequent requests from other requesters. In case the requester received a *RESP_REMOVED* or a *RESP_TAKEN* message, it aborts the reaction and returns the obtained molecules by enclosing them in *GIVE_UP* messages, which allows holders to give them to others.

Optimistic conflict resolution. Given the fact that the optimistic sub-protocol is designed to be executed by nodes in a highly *reactive* stage, there is no need for a strict conflict resolution policy. Instead, the node the request of which first reaches a holder obtains the desired molecule. However, the optimistic sub-protocol does not ensure that a reaction will be performed in case of a conflict. In the worst case, all attempts at fetching molecules might be aborted, for instance when three nodes try to fetch three molecules A,B and C, each node trying to get two of them in a cyclic way: a situation where $N1$ succeeds to fetch A, $N2$ succeeds to fetch B and $N3$ succeeds to fetch C, will inevitably lead to a general abort.

Algorithm 7: Optimistic sub-protocol: requester.

```
1  on event combination found
2      foreach molecule in combination do
3          dispatch FETCH(molecule)
4  on event response received
5      if response ≠ RESP_MOLECULE then
6          Abandon(combination)
7          return
8      add response.molecule to reaction_args
9      if all responses have arrived then
10         NotifyHolders(combination)
11         Reaction(reaction_args)
12 begin NotifyHolders(combination)
13     foreach molecule in combination do
14         dispatch REACTION(molecule)
15 begin Abandon(combination)
16     foreach molecule in combination do
17         dispatch GIVE_UP(molecule)
```

Algorithm 8: Optimistic sub-protocol: holder.

```
1  on event message received
2      if message = GIVE_UP then
3          molecule.state ⇐ free
4      else if message = REACTION then
5          remove molecule
6      else if message.molecule does not exist then
7          reply with RESP_REMOVED
8      else if molecule.state = taken then
9          reply with RESP_TAKEN
10     else
11         molecule.state ⇐ taken
12         reply with RESP_MOLECULE
```

3.3.3 Sub-protocol mixing

During its execution, a program typically passes through two different stages. The first one is the highly reactive stage, which is characterised by a high volume of possible concurrent reactions. In such a scenario, the use of the pessimistic sub-protocol would lead to superfluous network traffic, since the probability of a reaction's success is rather high. Thus, the optimistic approach is enough to deal with concurrent accesses to molecules. The second stage is the quiet stage, when there is a relatively small number of possible reactions. Since this entails highly probable conflicts between nodes, the pessimistic sub-protocol has to be employed in order to ensure the liveness of the system. Thus, the execution environment has to be able to adapt to changes and *switch* to the desired protocol accordingly. Moreover, these protocols have to be able to *coexist* in the same environment, as different nodes may act according to a different protocol at the same time.

Switching. Ideally, the execution environment could be perceived as a whole in which the switch happens unanimously and simultaneously. Obviously, a global view of the reaction potential cannot be maintained in a large-scale system. Instead, each node independently decides which sub-protocol to employ for each reaction. The decision is first based on a node's local success rate, denoted σ_{local}, computed on the basis of the success history of the last queries the node issued. In order not to base the decision only on its local

observations, a node also keeps track of local success rates of other nodes; each time a node receives a request or a reply message, the sender supplies it with its own current history-based success rate, stored into a list (of tunable size). Note that when there are multiple rules being executed, the node takes into account only the information relevant to the rule it is currently executing. We denote σ the overall success rate, computed as the weighted arithmetic mean of a node's local success rate and the ones collected from other nodes. Finally, the decision as to which protocol to employ depends on the rule a node wishes to execute. More specifically, it is determined by the arity of a rule, since the more molecules the rule needs, the harder it is to assure they will all be obtained. To grab r molecules, a node employs the optimistic sub-protocol if and only if $\sigma^r \geq s$, where r is the number of arguments the chosen rule has and s is a predefined threshold value. If the inequality is not satisfied, the node employs the pessimistic sub-protocol. Even though a more in-depth discussion about the value of the switch threshold falls out of the scope of the present work, we show its influence on the protocol's performance in Section 3.4.

Coexistence. Due to the locality of the decision of switching between sub-protocols, not all participants in the system will perform it in the exact same moment, leading to a state where some nodes try to grab the same molecules using different sub-protocols. In order to distinguish between *optimistic* and *pessimistic* requests, each requester incorporates a *request type* field into messages. Based on this field, the node holding the conflicting molecule gives priority to nodes employing the more conservative, pessimistic algorithm, so as not to jeopardise pessimistic protocol's liveness. More concretely, let O and P two nodes, respectively optimistic and pessimistic, trying to fetch the same molecule M from holder H. We can distinguish two cases: Firstly, if M receives O's *FETCH* message before P's *QUERY* message, O will receive the molecule and P will receive *RESP_TAKEN*, making it give up the reaction right away. In the inverse scenario, P is inserted in the list of requesters for M. If O's *FETCH* message reaches H before P's subsequent *COMMITMENT* message, O's message will be set back temporarily waiting for the end of the pessimistic round initiated by P. Doing so ensures that a pessimistic node (P or another node in M's requester list) will be able to grab the molecules it needs *eventually*, by preventing O (or any other optimistic node) to interfere in the pessimistic protocol.

3.3.4 Dormant nodes

During the quiet stage, the system might reach a point where $n \gg m$ (n denotes the number of nodes in the system, while m represents the number of molecules). In this extreme scenario, having many nodes represents a burden for the system, as most of the requests sent for molecules will ultimately be rejected by holders, elevating the network traffic without speeding up the progression of the computation. Thus we introduce the notion of *dormant pessimistic nodes* — nodes which are using the pessimistic sub-protocol to capture molecules, but do so less often than usual. When a node switches to the pessimistic sub-protocol, it starts counting the number of consecutively aborted reactions. Once this number reaches a threshold a, it puts itself to *sleep* for a predefined amount of time δ — it becomes a *dormant* node. It then *wakes up* and tries to capture another combination of molecules. In case it succeeds, it becomes an active pessimistic node again. Otherwise, it returns to the dormant state for another δ amount of time, and so forth. In order to avoid *massive awakenings* of nodes, *i.e.* the simultaneous resumption of activities of a large number of dormant nodes, we allow the actual amount of time a node spends as dormant to vary by a constant ϵ_δ from δ: before putting itself to sleep, a node randomly chooses the number of steps it is going to sleep for from the interval $[\delta - \epsilon_\delta, \delta + \epsilon_\delta]$. Dormant nodes do not put their entire execution on hold — they are still active in the system as molecule holders. Note that a discussion about concrete values of δ, ϵ_δ and a and their fine-tuning is here out of scope.

3.3.5 Multiple-rules programs

Thus far we focused on the protocol and its various aspects carrying the assumption that there is only one rule in the program. We need to extend the protocol to programs with multiple rules. As every node tries to carry out reactions, when multiple rules are present, each of them has to decide which of the rules it is

going to employ in a given cycle. Recall n denotes the number of nodes. Let $n_r \ll n$ be the number of rules. We refer to the rule being executed by a node at a given moment as its *active rule*. While deciding which of the sub-protocols to employ, a node should take into account exclusively the capture attempts made while executing the active rule. Thus, the calculation of the success rate is adapted as follows: each node maintains multiple success rates σ_i, one per rule i, and bases its switch decision on the relevant one. Note that this requires to expand the exchanged messages with one more field, namely the identifier of the rule for which the success rate is contained in the message.

Initial rule assignment. Nodes pick randomly one of the rules in the program with which to start the execution, except n_r nodes that are each permanently assigned a different rule. These *rule keepers* can be selected based on the hash identifiers assigned to the rules: a node is the rule keeper for a rule if the node identifier is numerically the closest to the rule's hash identifier. In case a node, according to the hash function, should be the rule keeper for more than one rule, it may delegate the responsibility for all but one of them to other randomly-selected nodes. Rule keepers try to execute their assigned rules all throughout the computation — they behave as if only one rule (the one they execute) is present in the system.

Changing the active rule. The reaction potential of a rule varies throughout the computation; depending on the state of the program at a given moment, more reactants may be present for one rule than another. Thus, nodes ought to be able to change their active rules during the execution based on the reaction potential of the rules. While a node is trying to obtain molecules using the optimistic sub-protocol, a change of the active rule is not being considered, as using this sub-protocol is an indicator of the active rule's high reaction potential. Hence, a change may occur when and only when a node is employing the pessimistic sub-protocol. A node changes its active rule if the following conditions are met:

1. the node is currently using the pessimistic sub-protocol;
2. the node did not succeed to perform a reaction in the previous cycle;
3. the success rate for the active rule observed by the node is the smallest amongst all rules' observed success rates.

The node then switches for the rule with the highest success rate known, If the current σ_i value of the newly selected active rule i permits the node to employ the optimistic protocol, it resets its grab history and sets σ_i to 1, since it means that its reaction potential is high, entailing a fair amount of reactions to be done with this rule. If even the new active rule's reaction potential is low, the node will become a dormant node immediately after changing its active rule. When it wakes up, it will start trying to apply the new active rule pessimistically.

3.3.6 Protocol's safety and liveness

A capture protocol needs to satisfy the two following properties:

- *safety*: a molecule is used in at most one reaction (we consider that every reaction consumes all of the molecules entering it);
- *liveness*: if at least one reaction is continuously possible, a reaction is performed in a finite time.

Theorem 3. *A molecule is consumed in at most one reaction.*

Proof. Let us concentrate first on the optimistic case (Algorithms 7 and 8) and assume by contradiction that a molecule is consumed twice. It follows that the same molecule was captured by at least two nodes. This in turn means that a *RESP_MOLECULE* message was sent by the holder to these two nodes for this molecule. This message can be sent only in case the molecule's state is *free*. A molecule cannot be in the *free* state

after being sent except in case the requester which obtained it finally abort the reaction and sends it back to the holder. It is consequently impossible for two nodes to obtain the same molecule at the same time.

Let us now consider the pessimistic case (Algorithms 5 and 6). Similarly assume by contradiction that two pessimistic nodes obtain the same molecule. It follows that they both entered the fetch phase, and that consequently, prior to that, they both received a *COMMITMENT* message sent by the holder. This means that both nodes were ranked first by the `SortRequesters()` function which can be true only if both nodes are not in the list of requesters for this molecule at the same time. Once one node enters the list, it can be removed from the list only upon reaction or if it abandons the reaction, making it impossible for the second node to be ranked first and thus obtain molecule if it was not either sent back to the holder or consumed in the reaction and removed on the holder's side. A contradiction.

Finally, there are two cases of conflict between the two protocols. Following our argument about the coexistence of the protocol, we see that when an optimistic request arrives before a pessimistic one, the pessimistic request is aborted because the molecule has already been reserved by the optimistic requester. On the other hand, if a pessimistic request arrives first, the optimistic request is aborted in favour of the pessimistic one.

□

Theorem 4. *If at least one reaction is continuously possible, one reaction is performed in a finite time.*

Proof. Let us first remark that under the assumptions that nodes are reliable and links are reliable FIFO channels, the protocol is deadlock-free. We can observe from algorithms that a requester will always get a response for each sent request. Based on the received responses, it will either perform the reaction or abort it and continue its execution, *i.e.,* retry to fetch molecules.

If at least reaction is possible, then it means that there exists a multiset M of molecules that is reactive, *i.e.,* matches the pattern and satisfies the condition of a rule.

Initially, nodes are all optimistic. In this configuration, either some node actually perform some reaction or all attempt at fetching molecules fail until all nodes switch for the pessimistic protocol (which necessarily arrives in a finite time in this case as optimistic nodes continuously try to fetch molecules and decreases their σ in case of failures.

Once all nodes are pessimistic, either part of the molecules in M are consumed, which means a reaction occurred, or M remains complete until the molecules in M constitutes the only reactants left in the solution. In the latter case, at least the rule keeper rk for this reaction will try to grab the molecules (we assume a perfect discovery protocol). Either it is alone trying to grab the molecules in which case it will actually succeed, or it is not. In this last case, either rk or another one succeeds to grab the molecules (due to the pessimistic resolution scheme) and perform the reaction.

□

3.4 Evaluation

Our protocol was simulated in order to better capture its performance. We developed a Python-based, discrete-time simulator, including a DHT layer performing the random dissemination of a set of molecules over the nodes, on top of which the capture protocol was implemented. The simulator adopted a discrete-time approach: any message issued at time t is received and processed by the destination node at time $t+1$. Also, after a reaction is performed or aborted at time t, the node tries to fetch another set of reactants. Unless otherwise noted, all presented experiments simulate a system of 250 nodes trying to execute a chemical program containing a solution with 15000 molecules.

The reactions' durations are assumed negligible, as our goal is to evaluate the capture protocol itself. For all of the simulations, we set the threshold between optimism and pessimism at $s = 0.7$, and the following values for constants related to dormant nodes: $a = 10$; $\delta = 20$; $\epsilon_\delta = 4$. Figures presented below show the values obtained by averaging result data over 50 runs. As the deviation for each simulation is negligible, we kept only the averaged values.

There are two sets of experiments. In the first one, we tested the protocol's behaviour, its performance and the network traffic generated by a simple program with a single rule. The second set of experiments examines the system's behaviour when executing programs with multiple rules. We simulated five different programs on top of the protocol. Note that the programs do not represent concrete implementations of applications since the actual reactions and their results are not taken into account. Rather, they were conceived in such a way as to examine the protocol in detail.

3.4.1 Single-rule experiments

In the first set of experiments we focused on the characteristics of the protocol itself, namely the performance and network overhead of each sub-protocol and the switch between them. Thus, the single-rule program was used throughout this set of experiments. It consists only of a simple rule which consumes two molecules without producing anything.

Experiment 1 (execution time). Firstly we evaluate separately the performance characteristics of both sub-protocols. Figure 3.3 shows the averaged number of reactions left to execute at each step, until inertia, using only the optimistic mode, only the pessimistic mode, and the complete protocol with switches between protocols (using $s = 0.7$), with and without the optimisation of dormant nodes, respectively. The *theoretic optimum* curve represents the amount of steps needed to complete the execution in an *ideal* distributed system, *i.e.,* with the highest possible parallelism degree and no conflicts during the whole computation: Considering that we need at least two steps to fetch molecules (one to request them and one to receive them), this hypothetical ideal system needs $2 * \frac{m}{nr}$ steps to complete, where m is the number of molecules, n the number of nodes and r the arity of the rule. This represents a lower bound on the number of steps, regardless of the model of computation, be it chemical or not. Note that a logarithmic scale is used for the number of reactions left. The figure shows that, when using only the *optimistic* protocol, there is a strong decline in the number of reactions left at the beginning of the computation, *i.e.,* when a lot of reactions are possible and thus there are only few conflicts. While this number declines, it gets harder for nodes to grab molecules, up to the point where the system is not able to move forward in the computation, as the few reactions left constantly generate conflicts and prevent termination. When the nodes are all *pessimistic*, there is a steady, linear decrease in the number of reactions left. The system is able to reach inertia, thanks to the liveness property of the pessimistic protocol. For most steps, the *mixed* curve traces the exact same path as the *optimistic* one, which means that during this period the nodes employ the optimistic sub-protocol. Towards the end, the system is able to quickly finish the execution as an aftermath of switching to the pessimistic sub-protocol. After the switch, it diverges from the optimistic curve to mimic the pessimistic one, exhibiting a 42% performance boost compared to the performance of the pessimistic sub-protocol. Comparing the theoretic optimum to our protocol, we notice an increase of 166% in the number of steps needed to reach inertia, which is not very conclusive, and can be seen as the cost of distributed coordination. Finally, as far as performance is concerned, including dormant nodes leads to similar results.

Experiment 2 (Switch threshold impact). Next, we want to assess the impact of the switch threshold s on the overall performance of the system. Figure 3.4 depicts, in the same logarithmic scale, the number of reactions left after each step for different threshold values, varying from 0.1 to 0.9. As suspected, the curves overlap during most steps, most nodes employing the optimistic sub-protocol. The first curve to diverge is the one where the switch threshold is set very high, to $s = 0.9$. Because the system depicted by that curve did not fully exploit the optimistic sub-protocol, it is the last to finish the execution. Conversely, a low value for s can lead to avoidable yet costly conflicts and impact negatively the performance of the execution. Although slightly, besides the one for $s = 0.9$, the curves start diverging at different moments, and, thus, complete the execution after a different number of steps. Figure 3.4 shows that, out of the five values tested for the switch threshold, $s = 0.7$ yields the best performance results in this particular scenario. Finding the optimal value for any application falls out of the scope of this study.

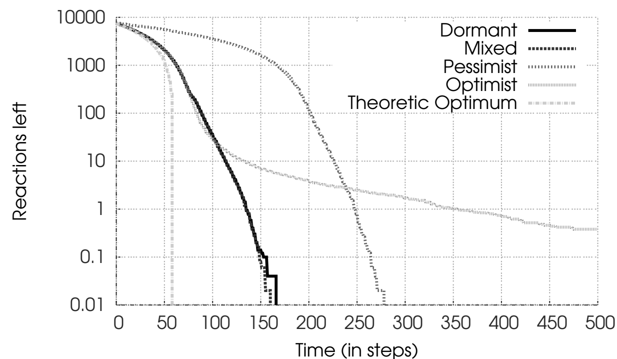


Figure 3.3: Performance of protocol's variants.

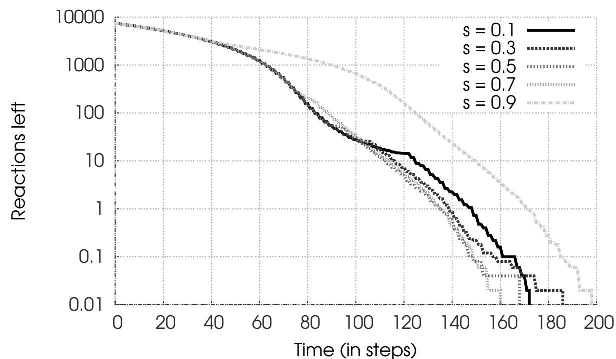


Figure 3.4: Influence of the switch threshold.

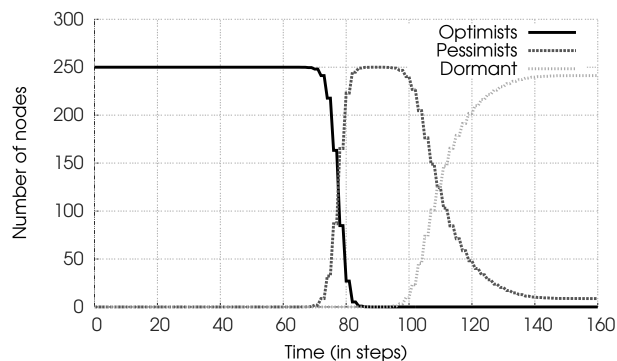


Figure 3.5: Evolution of the node population.

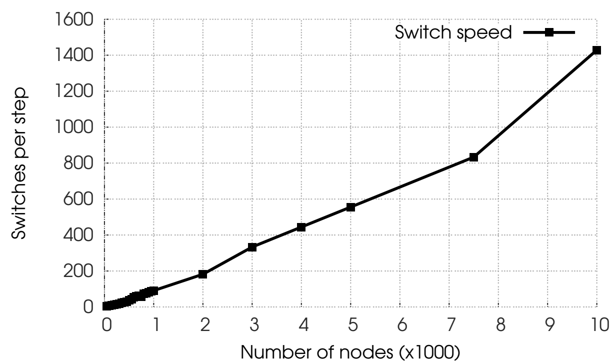


Figure 3.6: Switch speed.

Experiment 3 (Switch behaviour). Here we examine the behaviour of the process of switching from one protocol to the other. Figure 3.5 depicts the evolution of the number of nodes in each mode during the execution. We can see that, at the beginning of the execution, all of the nodes start grabbing molecules by using the optimistic sub-protocol. About half way through the execution, optimistic nodes start aborting more and more reactions, so they switch to the pessimistic sub-protocol. We observe that, thanks to the systematic exchanges of local σ values, nodes in the system reach a global consensus rather quickly — for a system with 250 nodes, at most 15 steps are needed for all of the nodes to switch to the pessimistic protocol. For the following 15 steps all of the nodes are active – and pessimistic – and try to capture molecules. However, as the concentration of molecules further drops, the number of dormant nodes increases, in this way reducing network traffic and allowing the still active nodes to capture the wanted molecules with more ease. Note that, while the overall number of dormant nodes increases, nodes wake up after a certain period and become pessimistic again. Figure 3.6 illustrates the number of nodes that switch from the optimistic to the pessimistic sub-protocol at each step during the transition period. One can observe that the more nodes in the system, the greater the number of nodes that switch per step. This behaviour comes from the fact that an increase in the number of nodes implies a greater accuracy of the system's state estimation σ as each node communicates with a wider spectre of nodes. This shows that the system, regardless of its size, can react quickly to changes, even though there is no global view of the situation.

Experiment 4 (communication costs). Next, we investigate the communication costs involved in the process. Figures 3.7, 3.8, 3.9 and 3.10 depict the number of messages sent per cycle in a system of 250 nodes. By *cycle*, we here means 12 simulation steps, as it is the lowest common multiple of 4 and 6; at most 4 steps are needed for the optimistic sub-protocol to complete, and at most 6 for the pessimistic sub-protocol.

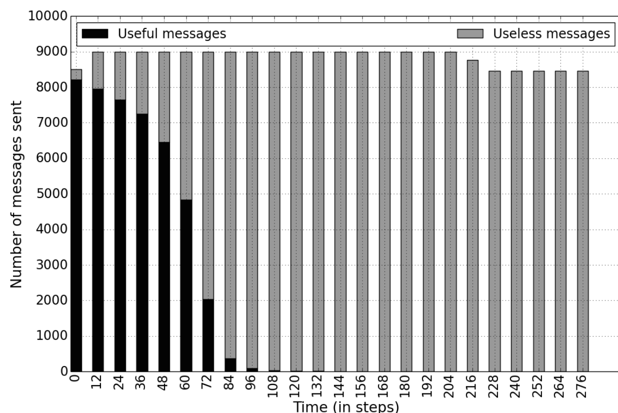


Figure 3.7: Number of messages when using the optimistic sub-protocol only.

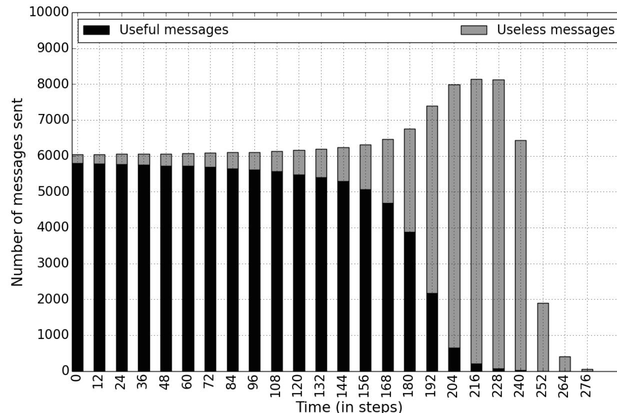


Figure 3.8: Number of messages when using the pessimistic sub-protocol only.

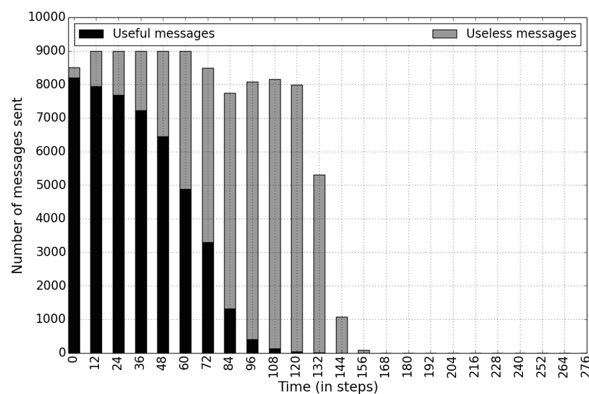


Figure 3.9: Number of messages when using the proposed protocol (without the dormant state).

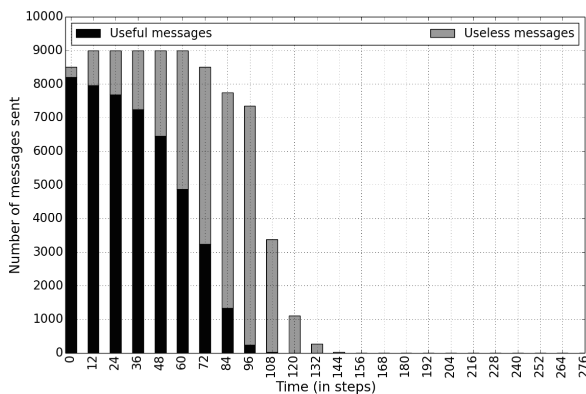


Figure 3.10: Number of messages when using the proposed protocol (with the dormant state).

The messages are classified into two categories: *useful* messages (ones which led to a reaction, in black) and *useless* messages (those which did not induce a reaction, in grey). When looking at the communication costs when only the optimistic sub-protocol is used (Figure 3.7), one can observe that a high volume of reactions is done in the beginning of the execution with a small percentage of conflicts, and thus a small amount of useless messages. However, as the execution progresses, the percentage of useful messages drops rapidly, while the total number of messages is kept high. Figure 3.8 shows that the exclusive use of the pessimistic sub-protocol consumes less messages, with the percentage of useful messages dropping steadily and slowly. At the same time, as there are less and less molecules in the system, the total number of messages slowly grows before peaking at around 8000 messages per cycle and then rapidly decreasing towards the end of the execution. When comparing Figure 3.9 to the previous two, we note again that the full protocol takes over the best properties of both of its sub-protocols: an elevated number of useful messages at optimistic times and a decrease in the total number of messages at pessimistic ones. A decrease of 30% in the number of messages can be observed when compared to the pessimistic sub-protocol alone. In addition, Figure 3.10 reveals that using the policy of dormant nodes further improves the scalability of the protocol, as it significantly reduces the total number of messages towards the end of the execution where there is the highest potential number of conflicts.

3.4.2 Multiple-rules experiments

In order to examine the behaviour of the rule-changing mechanism described earlier, we conducted a set of experiments with several multiple-rule programs. One of them was the simple split-merge workflow illustrated in Figure 3.11. It consists of four rules: $R0$, $R1$, $R2$ and $R3$. The rule $R0$ consumes two molecules of type $T0$ and produces two molecules: one of type $T1$, the other of type $T2$. These are used as input by $R1$ and $R2$, respectively. These rules can, thus, be run concurrently and independently of each other. $R1$ produces one molecule of type $T3$, while $R2$ produces one of type $T4$. Finally, their outputs are *merged* by the rule $R3$, which consumes one molecule per type — $T3$ and $T4$ — and produces no output.

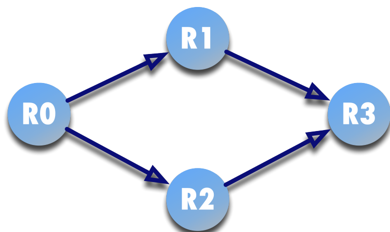


Figure 3.11: A rule-based workflow program.

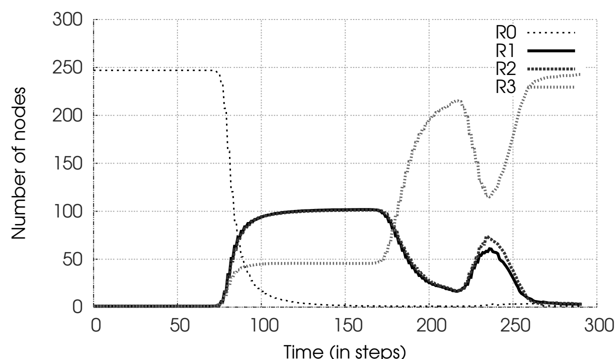


Figure 3.12: Number of nodes executing each rule.

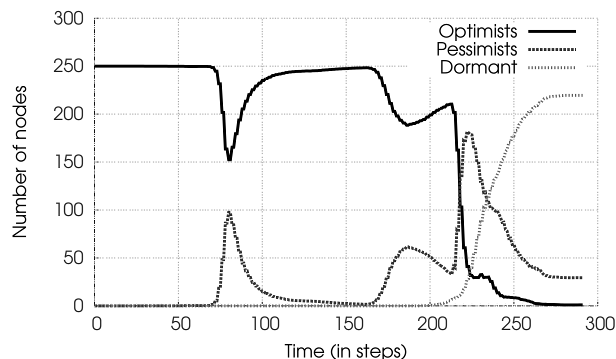


Figure 3.13: Number of nodes in each mode.

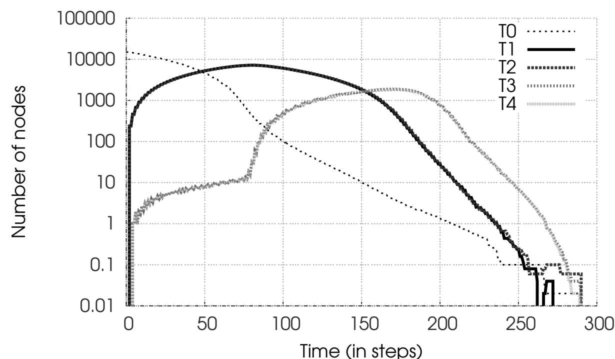


Figure 3.14: Number of molecules of each type (logarithmic scale).

The results of the execution of these programs are depicted in Figures 3.12—3.14. The rule-changing pattern reveals that the nodes first massively execute $R0$ until the concentration of $T0$-molecules drops below those of $T1$- and $T2$-molecules. The nodes then distribute themselves over the rules in such a way as to consume mostly these molecules — around 100 nodes per rule for $R1$ and $R2$. As they produce $T3$- and $T4$-molecules, about 50 nodes start executing the last rule, $R3$. As $T1$- and $T2$-molecules are consumed at a faster pace, nodes start abandoning $R1$ and $R2$ and pick $R3$. All the while, most of the nodes are employing the optimistic sub-protocol. Then, in between steps 200 and 250 about half of the nodes change back to $R1$ and $R2$ in order to consume the remaining $T1$- and $T2$-molecules, respectively, causing a decrease in the number of optimists, due to the globally low concentration of molecules. At the end of the execution almost all of the nodes use $R3$, most of them being dormant. Indeed, as there are only a few reactions left at that point, it is impossible for most of the nodes to perform reactions in spite of the fact that they have

correctly picked the rule to execute. This experiment shows that the rule-changing mechanism is able to follow the dependency patterns of a program with multiple rules. Moreover, we can see that during most of the execution the majority of nodes uses the optimistic sub-protocol, confirming that the protocol is able to adapt itself to the current situation in the system.

Experiments were conducted with other programs containing different dependency patterns. Please refer to the journal article [17] for the whole set of results we obtained.

3.5 Conclusion

This chapter proposed and validated the fundamental building block of our future distributed chemical machine: the atomic capture protocol. This protocol assumes molecules are already known by the nodes trying to capture them. A missing element towards a distributed chemical machine is the *discovery* mechanism. Another aspect ignored in this chapter is the detection of termination. While we studied it without any notion of decentralisation in Chapter 2, we now need to make the detection happen in distributed settings. The next chapter is about filling these two gaps, the implementation and experimental evaluation of a software prototype of a distributed chemical machine.

Chapter 4

Implementing Distributed Chemical Machines

This chapter describes two distributed chemical machines we have designed, developed and experimented in real settings, over the Grid'5000 platform. The first one is shaped as a hierarchical reactor *where reactions start at the leaves of a tree of compute nodes, and where inert subsolutions are then merged and reactivated as they move up the hierarchy until reaching the top level, the root of the tree. The second one fully decentralizes the discovery, capture and reactions of molecules, building a flat reactor. This second reactor supports the higher-order.*

Joint work with:

⋆ Marko Obrovac (PhD student, Inria)

4.1 The need for chemical run-time environments

Despite a large spectrum of applications (see [10] for an overview of them), the chemical model severely suffers from a lack of means for its execution over large-scale platforms and yet executing chemical programs is crucial on the road to its adoption in real contexts. This is particularly true in regard to its potential use in the specification of service coordination, to be detailed in Chapters 5 and 6. Without a *distributed chemical machine* able to execute the coordination, such works remain conceptual. In other words, we want to build a machine able to execute any chemical specification over large scale platforms, transparently for the programmer.

In [12], to introduce the philosophy of chemical programming, Banâtre and Le Métayer distinguished *logical* parallelism from *physical* parallelism:

> *Physical parallelism is related to the implementation; it corresponds to the distribution of tasks on several processors. By logical parallelism, we mean the possibility of describing a program as a composition of several independent tasks. Of course, a particular implementation can turn logical parallelism into physical parallelism, but these notions have very different natures: the former is a program-structuring tool, whereas the latter is an implementation technique.*

It appears that this distinction holds for distribution as well: reactions are implicitly local and autonomous. Time has come to offer an implementation to the chemical model, in other words, to turn its logical parallelism and distribution into physical ones. While the chemical model has been implemented over parallel machines (we review these works in the following), this has not been the case on more distributed platforms.

4.1.1 Physical parallelism

The execution of a chemical program carries three main problems: (i) finding reactants, *i.e.*, retrieving combinations of molecules that satisfy some reaction rule's conditions, (ii) performing reactions, *i.e.*, deleting the molecules found in step (i), producing the result of the reaction, and inserting it in the solution, and (iii) detecting inertia, *i.e.* detect the fact that no more combinations of remaining molecules satisfy any reaction rule's condition. An execution machine performing these steps exists for a single processor [83] for HOCL.

A series of works starting from the end of the nineteen-eighties to the middle of the nineteen-nineties targeted the implementation of GAMMA over specific parallel machines: In the founding article proposing GAMMA [9], the method proposed for implementation relies on molecules travelling from processor to processor, processors being interconnected in a vector, each processor having two neighbours. Molecules then move along the vector either until they react or until they have returned to their starting point. This algorithm was implemented on an iPSC hypercube with 16 processors. Let us also mention the work of Gladitz and Kuchen around the implementation of the map and *fold* (also known as *reduce*) operations over *bags* (*i.e.,* multisets). They implemented them over a Sequent Symmetry S16 composed of 6 processors [63].

On the SIMD machines side, a similar implementation algorithm, based on the odd-even transition sort has been implemented on a Connection Machine [35]. The Connection Machine is composed of $64K$ processors interconnected in a 12-dimensional hypercube in which each node is made of 16 fully connected processors equipped with $32k$ of memory and a 1-bit wide ALU. The same kind of work has been conducted with the MasPar MP-1 with 8192 4-bit processors interconnected as a 64×128 square array, using the fold-over operation: Molecules are placed over a strip. At each step, the elements in the upper segment of the strip are compared in parallel to those in the lower segment.

The topic of implementing the chemical model on top of parallel platforms seems to have been dormant until Lin *et al.* shown that GAMMA can be used to model programs executable on a cluster exploiting GPU computing power [66]. Although all these works present significant execution time speed-ups, their execution is strongly hardware-dependent, and conceived for specific tightly-coupled architectures: parallel machines and GPUs. Our goal is to explore the feasibility of a generic large-scale chemical platform, to be deployed over potentially geographically dispersed, heterogeneous nodes.

4.1.2 Physical distribution

The rest of the chapter presents our work around turning the logical distribution of the chemical model into a physical one. It is divided into two parts. The first part, in Section 4.2, presents the design, implementation and experimental evaluation of a *hierarchical* reactor. In this reactor, there is no need for a discovery protocol: Each node is initially given a subset of molecules and tries to reduce this subsolution until inertia is locally reached. At that point, it sends its inert subsolution to its parent in the hierarchy. Once a node receives the inert solutions from all its children, it merges these subsolutions and restart reducing the solution. This process is carried out in a distributed asynchronous fashion throughout the hierarchy until all the molecules that survived the repeated reductions reach the top of the hierarchy for a final reduction attempt.

The second, *flat* reactor, detailed in Section 4.3, does not require to define a static hierarchy. It operates in a fully decentralized manner where each node constantly tries to i) discover reactants, ii) capture discovered reactants and iii) perform a reaction on reactants captured.

4.2 A hierarchical reactor

This section focus on our first design of a distributed chemical platform. We opted for a hierarchical approach, where the solution is initially dispatched equally over a set of compute nodes. As soon as a node declared inertia over its own portion, it will merge it with the inert portion of another node. This parallel reduce-and-merge process will continue until having only one node gathered the whole solution to apply the last phase of reduction locally and obtain the final inert solution, *i.e.,* the result of the execution. In the following of this section, we detail the design, implementation and experimental validation of such a platform. To find reactants and detect inertia, which is the key element of the reactor, we first provide a simple *brute-force*

reduction algorithm, which is sub-optimal in terms of how many reaction tests are done. In other words, it tests more molecule combinations than actually required to detect inertia. We also discuss how to inject load balancing within such an architecture, by presenting a scheme to reorganise the execution flow.

4.2.1 Overview

The hierarchical reactor is illustrated in Figure 4.1.

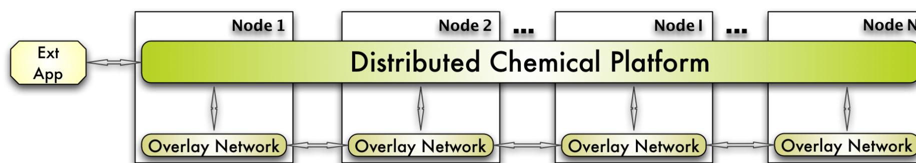


Figure 4.1: The distributed chemical platform.

In order to abstract out the heterogeneous, dynamic and large-scale nature of targeted platforms and secure independence from the underlying environment, the reactor relies on a distributed hash table (DHT) to interconnect the nodes involved in the reactor. The external application sending its program to the chemical runtime platform can contact any of the DHT nodes acting as an entry point to the platform, thus avoiding a potential bottleneck due to the uniqueness of an entry point. Note that the prototype to be described relies on the Pastry [88] DHT. Any DHT could fill this role. Once a node has been contacted by the external application, it becomes the *source* for this particular execution and the root of the execution tree to be created. At the end of the execution, it will send the inert solution back to the application. First, data and rules received by the source have to be sent to the nodes. Using Pastry's hash function, the source distribute the molecules uniformly on the nodes. Molecules are routed concurrently according to Pastry's routing scheme, in $O(\log n)$ hops [88]. In the course of the routing process, the path of each molecule is traced by intermediary nodes, called *forwarders*, from the source node to a molecule's destination node, referred to as a *worker*. By passing on molecules, forwarders maintain a *local state* (in addition to the Pastry's routing table) containing the set of nodes to which they forwarded molecules. Note that forwarders, together with the source node, can be workers as well, the number of molecules being typically much higher than the number of nodes. Finally, the source node spreads one final message mr containing the rules to execute down the graph thus created. During the spreading of mr, the graph, which might contain cycles (as a node may have several parents) is made a tree, each node keeping as parent only the first node to transmit it the mr message. This tree creation is illustrated in Figure 4.2 with a 16-node DHT where the source is node 0. In Figure 4.2(b), molecules with keys 1, 3, 5 and 7 are distributed, thus starting the creation of the tree. Once all of the molecules have reached their destinations, the complete execution tree is in place (Figure 4.2c).

Once mr is received, the first nodes to act are the leaves of the tree. Each of them sends its inert local solution to the respective parent. The parents then add them to their own and continue the computation. Only when the parent has received all of its children's solutions and when its local solution is inert, the process continues with the parent transferring its local solution to its parent, and so forth until all of the inert local solutions reach the source node, which delivers the global solution after executing it until inertia.

4.2.2 Efficient condition checking and inertia detection

We now discuss how to make sure the amount of work done throughout the distributed process of checking conditions and detecting inertia is *efficient*. In particular, we need to ensure that, when molecules move up in the hierarchy, no useless work is done. More precisely, we need to be sure that, when inertia is reached, it is detected using an *optimal* number of tests on the molecule, *i.e.,* that every combination of molecules is tested against the condition once and only once. For the sake of discussion, and for an accurate study of the issue, we present two algorithms distributing the task of trying every possible combination of molecules.

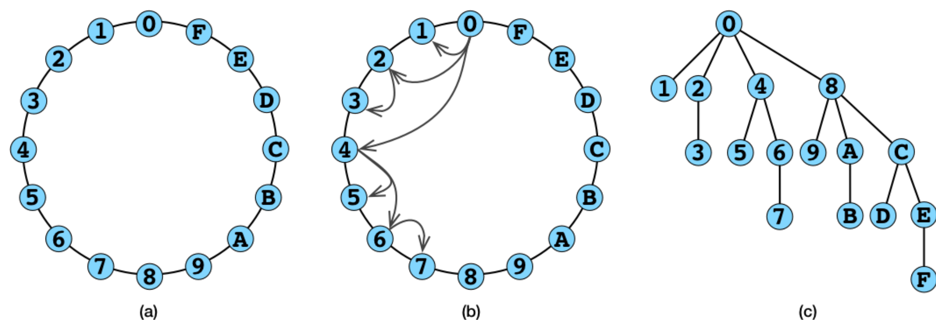


Figure 4.2: Execution example: (a) the original ring; (b) molecule dissemination and tree creation; (c) the execution tree.

The first one, based on a repeated brute-force mechanism, is intuitive but sub-optimal, highlighting the fact that, if one is careless, many unnecessary tests can be done, increasing the complexity of an already hard task. The second one, referred to as *BucketSolver*, is shown to be optimal in terms of number of combination tests.

Brute-force approach

An intuitive way to perform local reactions can be the following: a node first reduces its own local solution to inertia. Upon receipt of the inert solution from one of its children, it merges both solutions and restart the process leading again to inertia. If one node has got g children, there are exactly $g + 1$ reductions before the node sends its solution to its own parent. It is quite easy to see that, following such an approach leads to a sub-optimal number of checks: upon merging, a node, which did not keep track of already tested combinations will retest them in addition to new possible combinations. Note finally that this extra cost depends on the number of children and more generally on the number of nodes in the tree, thus potentially significantly limiting its scalability. The extra cost is discussed in more detail in [79].

Bucket Solver

When a node transfers its local solution to its parent, the parent is sure that all of the combinations in the node's local solution have already been tried. The parent does not need to know which combinations have been checked: it only knows the set of molecules they derive from. Thus, we create *buckets* into which we put sets of molecules the combinations of which have already been tried.

When a node originally receives molecules from the source, it puts them in its own bucket and reduces the subsolution it contains. The following local reduction cycles on this node will check only inter-bucket combinations — where elements are taken from different buckets. During each cycle, a node chooses two buckets (say a and b) among the buckets it currently manages. Then, if the rule it applies requires r molecules, it picks j, $0 < j < r$, elements from bucket a and $r - j$ elements from bucket b. If the combination is evaluated positively, the elements are removed from their respective buckets and once the reaction has been carried out, each resulting molecule is placed in a new, separate bucket.

Once two buckets' inter-combinations have been checked, they are *fused* — the molecules from one bucket are put into the other and the now empty bucket is deleted. The process is repeated until the buckets from all children have been received and there is only one bucket left. Following this logic, the solution, be it local or global, is declared to be inert once there is only one bucket left on the node or in the system, respectively.

Consider the example illustrated in Figure 4.3. Imagine a node checked two molecules, a_1 and a_2, which now reside in bucket a. The node then receives an inert local solution from one of its children and creates a new bucket, bucket b (Figure 4.3(I)). Now, it checks all of the combinations except those of elements residing in the same bucket. In this example, the node checks the following combinations: (a_1, b_1), (a_1, b_2), (a_2, b_1)

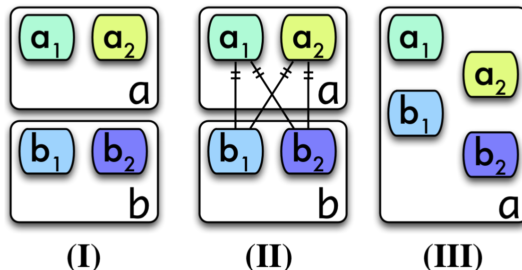


Figure 4.3: (I) Buckets to check, (II) buckets being checked, (III) combined buckets.

and (a_2, b_2) (Figure 4.3(II)). Note that the combinations (a_1, a_2) and (b_1, b_2) are not checked, for they have been previously examined. Finally, presuming no reaction took place, the two buckets are combined into one containing all of the elements (Figure 4.3(III)).

This algorithm is valid for an arbitrary number of reactants to find, *i.e.*, for an arbitrary size of the left part of rules, since at least one element per bucket must be picked. In the case of a reaction rule needing only one molecule to be triggered, then condition checks only need to be performed on the molecules received at the time of the initial dissemination. The ones transmitted from children are only forwarded to parents, as they were already checked against the reaction condition by their initial worker, some leaf node in the sub-tree. Thus, every molecule is checked once and only once.

The BucketSolver algorithm provides *inertia detection* (all of the combinations will be examined by the reaction condition) while being *optimal* (every combination will be checked only once).

4.2.3 Tree reorganisation

One problem not addressed thus far is the shape of the hierarchy. The execution tree creation heavily depends on the underlying overlay network's properties, in particular its topology (the term *neighbour* having different meanings in different network organisation schemes), its routing algorithm and also the hash function. Moreover, given the tree-shaped execution flow, the root node may, depending on the type of problems being solved, face a growing probability of overload, in particular in problems where the number of molecules does not decrease over time. Thus, the load may be poorly balanced over the nodes of the tree, which could lead to performance degradation and render the algorithms for testing reactants prone to communication and computation bottlenecks. To avoid these problems, we propose a tree reorganisation scheme, where each group of children of a given node is organised into a new sub-tree having this node as root.

Reorganisation scheme. The basic idea consists in modifying a node's local state, namely its children and parent entries, so as to limit the number of children per node. Consequently, once a node reaches inertia, it will send the local subsolution to a parent, not chosen through the initial DHT-based dissemination process, but by the reorganisation scheme. This scheme scales gracefully because changes are propagated top-down from the source and applied locally to nodes. Moreover, it is a cost-effective scheme in the sense that no extra messages are needed for said propagation. The changes are operated as the rules are disseminated together with the rules in the mr message, which is expanded to carry additional information, namely the maximum number of children allowed per node, denoted g_r, and the list of nodes which will be children of a given receiver of mr. g_r, $2 \leq g_r < g$, is the reorganisation scheme's input parameter and is set by the user. The process starts at the source node. The children in its local state are sorted based on their indices and are then split in g_r groups. In each group, one child (for instance the one having the highest index) is elected as a *group leader* and remains the source node's child. The rest of the group is removed from the source's local state and referenced in the mr message, sent to the group leader. Upon receipt, the group leader will set the sender as its parent, add the list of nodes in mr to its local state and repeat the grouping at its

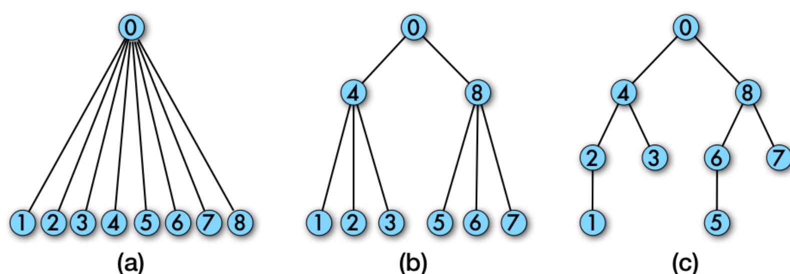


Figure 4.4: Tree reorganisation example: (a) initial state, (b) first level established, (c) tree completely reorganised

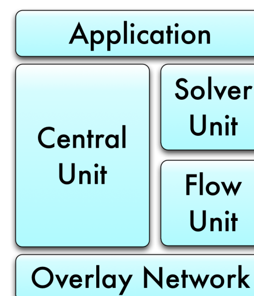


Figure 4.5: Software architecture of the hierarchical reactor prototype.

level, and so forth until reaching the bottom of the tree. The reorganisation stops once all of the nodes have received an *mr* message, which marks the beginning of the execution (using the BucketSolver algorithm).

Example. Let us demonstrate the reorganisation process on a simple example. Figure 4.4a depicts the execution tree built after disseminating molecules with ids 0 through 8 in a 9-node system. Let node 0 be the source node and let $g_r = 2$. Node 0 splits its children list in two groups. It keeps node 4 from the first group (1-4) and node 8 from the second (5-8). It then sends one *mr* message including the list of non-leader nodes of the first group, *i.e.*, the list $1, 2, 3$, to their group leader — node 4. Another *mr* message containing the list $5, 6, 7$, is sent to node 8. Node 0 can now starts its execution. Concurrently, nodes 4 and 8 update their local states by setting $parent_id = 0$ and copying the respective lists they receive lists to their respective children entries (Figure 4.4b). Node 4 now splits its children list into two groups ($< 1, 2 >$ and $< 3 >$) and elects the leaders (2 and 3). Two *mr* messages are then constructed; the first one containing only 1 in its list is sent to node 2, while the other is sent to node 3 with an empty list. Node 4 now starts the execution. Nodes 2 and 3 set their parent to 4. Node 3 starts reducing its local solution, while node 2 sends *mr* to node 1 which sets its parent and starts the execution. The exact same effect is achieved on the right side of tree: node 8 splits its children list into two groups, sending 5 to node 6 and no list to node 7 (Figure 4.4c).

4.2.4 Prototype

To better capture the viability and performance of the hierarchical reactor, a software prototype of the architecture and algorithms presented above was developed[1]. Its components are depicted in Figure 4.5. The overlay network was implemented with FreePastry [2], an implementation developed by the original authors of Pastry. Its facilities are used by the two units directly above it — the central unit and the flow unit, both present on every node.

Central Unit. The central unit is the entry point of the solver, it is in charge of accepting and taking over the requests originating from external applications. It hashes the molecules and send them over to FreePastry that will actually route them to appropriate nodes. It also initiates the dissemination of the rules by forwarding them to the flow unit which will actually perform the multicast. Once global inertia has been achieved, it is informed by the flow unit and delivers the solution to the application.

Flow Unit. Meanwhile, each flow unit monitors the traffic in the DHT and builds its state, *i.e.,* its view of the execution tree, based on routing decisions taken by the overlay network. Once the dissemination process has completed, the flow unit holds the list of its direct tree descendants and the parent's identity. As

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

specified before, this list is used at execution time as a synchronisation barrier: the flow unit forbids sending the local inert solution to the parent until all of its children's results have been received. The flow unit is also naturally responsible for the tree reorganisation (during the rules dissemination): it elects the group leaders, removes the rest from its children list and passes their identities on to the new group leaders.

Solver Unit. The key role in the execution is played by the solver unit, which is the implementation of the algorithms proposed in Section 4.2.2. The unit has two different implementations, one for each of the two condition checking algorithms. After the dissemination process, the central unit triggers its execution of the local solution. Once inert, the solution is transferred to the flow unit. Also, when a result from a child is received, the flow unit hands it over to the solver unit, triggering a new execution cycle. The BucketSolver algorithm has been implemented in the solver unit in a multithreaded version: the well-defined bucket boundaries imply that two disjoint groups of buckets, each containing at least two bucket, can be processed independently one from the other. As a result, the solver unit implements a thread pool which schedules the execution of groups of buckets on free threads in the pool as long as the local solution is not inert, *i.e.*, as long as there are at least two buckets left to fuse. The synchronization this parallelization requires is very light. Note that such a mechanism cannot be applied to the brute-force algorithm that has no *recollection* of the combinations previously checked. Parallelizing the brute-force approach would call for a more complex synchronization, as two threads could try combinations containing the same molecule, leading to inevitably poor speed-ups.

4.2.5 Experimental evaluation

This section presents the results of an experimental evaluation of the prototype of the hierarchical reactor presented in Section 4.2.4. The goal of this experimental campaign is fourfold: (i) examine the viability of the architecture, (ii) compare the reactant finding algorithms, (iii) look at the network traffic generated and (iv) test the tree reorganisation scheme laid out in Section 4.2.3. The experiments were conducted on the French nation-wide Grid'5000 [27] computation platform, connecting nine geographical-distant sites through the RENATER network[2], which offers a 10 GBit/s communication channel between the sites. We varied the number of chemical nodes taking part in the reactor from 100 to 1000 scattered randomly across the nine sites, each Grid'5000 node hosting up to its number of cores as chemical nodes. The results represent values averaged over 6 runs. The configuration of the overlay network was changed upon each run. The prototype was tested on two chemical programs presented hereafter.

Test Programs

The evaluation of the proposed architecture and algorithms was conducted using two chemical programs. While the programs tested might seem rather simple, their interest stands in that they belong to a broader class of programs having a specific behaviour at run time. Even though their expression is quite simple (thanks to the uncluttered style of the chemical model), their runtime's complexity is similar to many other chemical programs with a more complex logic. Put simply, what matters here is the complexity of finding reactants over time. The first class of applications has a complexity that decreases over time, while the second exhibits a constant complexity over time.

Programs with a decreasing density. A large collection of real-world applications solve problems the complexity of which gradually decreases as computation progresses. In our experiments we chose to represent this class of applications with the *getmax* program discussed in Chapter 2. *getmax* is a so-called *reducer* rule since each reaction decreases the total number of molecules, and consequently the density of reactants (and reactions). This is a typical scenario for many data-processing applications where the amount of data to be processed diminishes over time. In the experiments, the *getmax* program was performed over a solution initially containing 50,000 molecules. Data processing itself was simulated through a local one-second pause after each reaction.

[2] `http://www.renater.fr`

Programs with a constant density. Our second test program is a program made of one rule acting on molecules composed of two integers — an index and a value associated with it. The goal of the program is to sort the value through the indices. The *sort* rule is given below:

$$\begin{aligned}\textbf{let}\ \ sort\ &= \textbf{replace}\ \{x.i, x.v\}, \{y.i, y.v\}\\ &\quad \textbf{by}\ \{x.i, y.v\}, \{y.i, x.v\}\\ &\quad \textbf{if}\ (x.i > y.i \wedge x.v < y.v) \vee (x.i < y.i \wedge x.v > y.v)\end{aligned}$$

The rule consumes two molecules if they are not already sorted in ascending order. Two new molecules are then created, holding the same indices as the original ones, but with swapped values. Although simple, this program is relevant for our experiments as it keeps the number of molecules in the solution constant — every time two molecules are consumed, two new are created. This means that the complexity of finding reactants remains constant throughout the execution, which allows us to better compare BucketSolver to the naive approach. Also, keeping constant the number of molecules and thus the amount of data, gives unbiased insight into possible bottlenecks and performance degradations.

Results

The four metrics evaluated so as to capture the behaviour of the hierarchical reactor prototype and verify the properties we devised before are: (i) execution speed-up, (ii) inertia detection complexity, (iii) communication cost and (iv) effect of tree reorganisation.

Execution speed-up. Firstly, we examined the speed-up that can be obtained using the hierarchical reactor. Figures 4.6 and 4.7 show the speed-up in execution time achieved by comparison to the execution time of a single executor. Using *getmax* (see Figure 4.6), we observe that both reactant finding algorithms achieve significant speed-ups, linearly growing with the number of nodes. However, while the brute-force reaches a maximum speed-up of 150, BucketSolver is able to reduce the execution time by a factor of up to 900. Although both algorithms perform the same number of reactions, when using BucketSolver, the runtime benefits from the implementation's optimisation of multi-threading, which allows it to perform multiple reactions at a time. On the other hand, Figure 4.7 shows that in the case of the *sort* test program distribution of the execution does not induce a significant speed-up. Concretely, the brute-force algorithm is barely able to match a single instance's execution time regardless of the number of nodes, while BucketSolver achieves speed-ups in the range $5-8$. In contrast to *getmax* where parents in the tree have got less work to do after receiving the results from their children, a node receives the same amount of molecules they forwarded to their children during the initial dissemination. Thus, even though the total number of molecules in the system is constant, the complexity of the program per active node increases over time. Consequently, parents have got to wait longer and longer on their children's responses as the computation is moving up in the tree, increasing the global execution time. BucketSolver still performs better due to its optimality and ability to exploit multi-threading.

Inertia Detection. Next, we wanted to experimentally verify our finding that the BucketSolver is optimal in terms of number of tests when the solution is inert. We did so using the *sort* program on an inert solution, *i.e.,* an already sorted one. As expected, our observations — results are depicted in Figures 4.8 and 4.9 — were that the brute-force algorithm induced a systematically slower execution (by a factor between 4 and 5 compared to BucketSolver). The exact extra-cost over BucketSolver yet fluctuated as a result of building different execution trees for each run, due to the different overlay network's initial configuration. Also, as predicted, the total amount of combinations tested by BucketSolver matches that of a single executor, while that of the brute-force algorithm fluctuates again wildly. Finally, as already suggested by our experimentations about the speed-up, the number of nodes in the case of the *sort* program does not greatly reduce the execution time, suggesting that nodes spend a lot of time waiting for results from nodes in their subtree.

Let us briefly mention our experimental findings on communication overhead and the impact of the tree reorganisation. These aspects are reviewed in more detail in [79]. We observed that the communication overhead is linear in the number of nodes and depends on the type of programs executed. In particular,

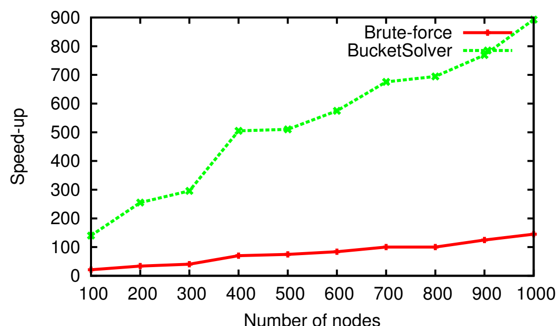


Figure 4.6: Speed-up in execution time for the *getmax* program.

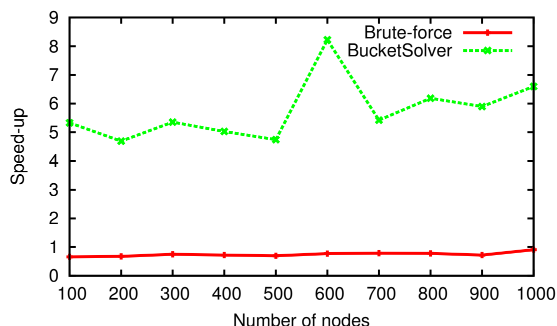


Figure 4.7: Speed-up in execution time for the *sort* program.

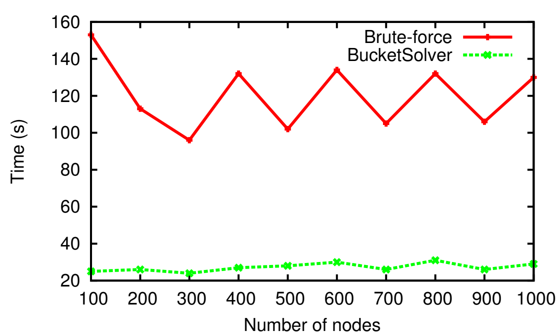


Figure 4.8: Execution time on an inert solution (*sort* program).

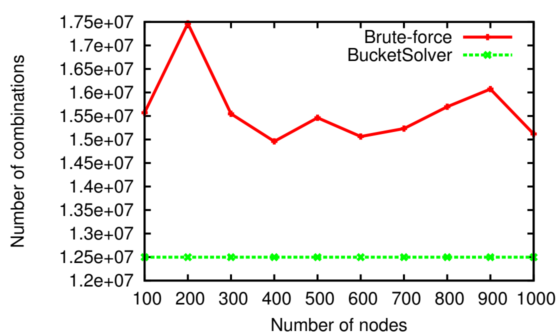


Figure 4.9: Number of checks done on an inert solution (*sort* program).

the overhead of adding new nodes is smaller for *getmax* since its complexity decreases over time. For both programs, the overhead per node decreases, showing that no bottleneck due to traffic is to be expected when the platform grows. Finally, our experiments involving the tree organization led us to the conclusion that the improvement it brings in terms of load balancing is significant, in particular on the source node: the percentage of reactions done by the source for the *sort* program with 400 nodes is reduced from 65 to 25. Still, it does not bring a significant speed-up as long as the upper part of the hierarchy does not constitute a bottleneck. Also, while the reorganization leads to a better load balancing, it also deepens the tree, making path from the leaves to the source longer.

4.3 A fully-decentralised higher-order reactor

The hierarchical reactor presented was the result of building a proof of concept of a decentralised chemical machine. While it constituted a significant first step, several issues were still to be solved. Firstly, decentralisation is still limited in the hierarchical reactor: the load on the nodes is by construction, not uniformly dispersed. In particular, a higher node in the tree is likely to experience a higher load. In other words, the hierarchical reactor suffers from its hierarchical shape. Even if all nodes apply the same algorithm, they might end up having a very different workload. Secondly, the hierarchical reactor does not guide the search. The molecules are randomly dispatched and moved when they cannot react any more without being fused with other buckets. The algorithm does not *search* for reactants but rather checks in parallel all the combinations in an exhaustive manner. By some simple pre-analysis of the rule, guiding the search towards molecules that has an actual chance of implying a reaction may significantly reduce the global complexity

of the reactor. Thirdly, the hierarchical reactor does not take higher-order into account. Modifying the set of enable rules dynamically is not possible. Actually, it would require a significant cost for synchronization: If some node applies a higher-order rule removing a rule, no other node should apply it anymore, calling for mechanisms to inform every node of this change, such as another broadcast.

The *flat* reactor presented in the following addresses these problems so as to offer a fully-decentralised platform to execute higher-order chemical programs. Although, as the hierarchical reactor, it is based on a DHT to abstract out the details of the infrastructure, it does not build a rigid hierarchical structure to dispatch and gather molecules. Instead, it builds a second DHT layer on top of the first one to store meta-data about the molecules and their state of *reactiveness*. Based on a specific way to order them, searching for reactants and detecting inertia is achieved through exploring this second layer. Once reactants are found, they still need to be *acquired*. This is done through the use of the adaptive capture protocol detailed in Chapter 3. To make it possible to support higher-order rules, mechanisms to track a rule's existence are included.

The rest of the current section is structured as follows: Section 4.3.1 presents the general execution scheme followed by a node in the flat reactor. Section 4.3.2 presents in detail how molecules are retrieved and how inertia is detected using the two-layer DHT. Section 4.3.3 presents the mechanisms enhancing the reactor with higher order. Finally the flat reactor prototype we developed is described in Section 4.3.4, and experimental results, again obtained by large deployments over the Grid'5000 platform are discussed in Section 4.3.5.

4.3.1 General execution scheme

The general idea of the underlying layers of the flat reactor is not different from the hierarchical reactor depicted in Figure 4.1. A set of nodes are interconnected into a network, each of them being equipped with the same engine. The engine of the flat reactor is just different from the hierarchical one. We assume that molecules are, as within the hierarchical reactor, randomly dispatched over the nodes. The *source* still refers to the node initially contacted by the application that will send back the result at the end of the computation. We also assume again that the rules of the program are present in every node before nodes start trying to make reactions.

A difference with the hierarchical reactor is that, to help discovering reactants, a second DHT layer containing information about molecules, referred to as *meta-molecules* in the following, is distributed over the nodes of the platform in an order-preserving manner. Each molecule in the first layer of the DHT is associated with a unique meta-molecule in the second layer. This allows molecules with values within a given range to be searched more efficiently (for instance an integer greater than 3). As we will detail later in Section 4.3.2, each meta-molecule has a state which is either *free* meaning it may be used for a reaction, or *inert*, which denotes that a node tried to make it react but was unable to find other reactants so as to build a reactive combination containing it.

Algorithm 9: 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)
```

Once molecules and rules have been dispatched, the main execution loop of every node is described in Algorithm 9, in a simplified form as it considers only one rule acting on pairs of molecules. The algorithm can still be easily extended to multiple rules with arities greater than 2. The loop consists in three steps: *(i)* getting a random meta-molecule or detecting inertia (lines 2—4), *(ii)* finding a candidate molecule it can react with (lines 5—9) and *(iii)* atomically grabbing the corresponding molecules and performing the reaction (lines 10—14). Let us briefly review these three steps. Step 1 (lines 2—4) initiates the loop: a node first tries to obtain a random meta-molecule, the state of which is *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 that, previously, another node could not find any candidate to react with the currently present molecules, implying their states were set to *inert*. This signals to the requesting node that inertia has been reached. It then stops executing the main loop (line 4).

Step 2 (lines 5—9) is triggered when a free meta-molecule has been obtained. The node now asks the system to find a second suitable meta-molecule by supplying the meta-molecule found in step 1 and the rule which needs to be applied on the molecules to the `find_candidate` routine (line 5). This routine searches for a meta-molecule to match the provided rule's pattern and reaction condition when combined with the first molecule obtained. Note that, in this routine, both *free* and *inert* molecules are checked. If no suitable candidate is found, then the state of the first meta-molecule obtained is changed to *inert* and stored back in the second DHT layer.

Step 3 (lines 10—14) concludes an execution loop iteration. The node tries to grab atomically the molecules described by the previously obtained meta-molecules. If the `grab_molecules` routine succeeds, it ensures no other node will obtain these molecules, making it possible to trigger the reaction, after which the meta-molecules describing the newly created molecules are produced. The new molecules and their corresponding meta-molecules are then sent to their respective nodes based on their hash identifiers for storage. It is important to note that, 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 after that, the meta-molecules corresponding to the consumed molecules are removed. This order ensures that no node will be able to declare inertia before it is actually reached.

4.3.2 Molecule searching and inertia detection

Let us have a more precise look on the way things are done to search molecules. Size of molecules can vary a lot, potentially provoking network congestion if moved frequently. In order to reduce superfluous network traffic, on top of the Pastry ring (the *uniform layer*) we place a second one containing *meta-molecules* positioned in an order-preserving manner (the *order-preserving layer*). The logical placement of the two layers is visible on Figure 4.10. Since both layers share the same key space, each node is in charge of both molecules and meta-molecules residing within its responsibility area.

The first layer uses a uniform hash function to spread the molecules uniformly over the nodes. The source node initially scatters the data molecules across the system according to their hash values. As for the hierarchical reactor, rules may be broadcast using the tree rooted at the source created by the disseminations of molecules. Yet the role of the tree is limited to the initial dissemination of rules in the flat reactor and the final gathering of remaining molecules to be sent over to the application. The first layer is used during Step 3: Once a combination of reactants has been found, the molecules are sent to the node that will host the reaction. In other words, they are effectively removed from the layer. After the reaction, the newly created molecules are hashed with the uniform hash function and also put into this layer.

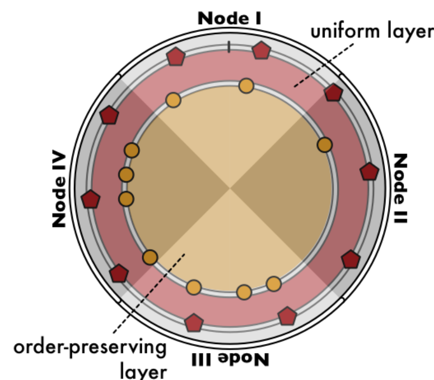


Figure 4.10: Double layer for a 4-node DHT: the key space of the uniform layer storing molecules (red pentagons) coincides with that of the order-preserving layer (yellow circles). The alternating greyed and not greyed regions designate responsibility areas of different nodes.

The second DHT layer physically matches the first one: both use the same key space and nodes keep their identifiers as well as their routing tables. Still, it stores meta-molecules — objects which describe the molecules. Each of them carries four pieces of information: the hash identifier of the molecule in the first layer, its type, its cardinal position in the molecule's type's total order and its state — *free* or *inert*.

In order to efficiently locate *free* meta-molecules, the key space is split into two equal parts: the one containing only *free* meta-molecules, within the range $[0, \frac{ks}{2} - 1]$, and the other consisting of only *inert* meta-

molecules, within the range $[\frac{ks}{2}, ks-1]$, where ks is the size of the key space. This means that once the state of a molecule has been changed, its meta-molecule's identifier changes as well, as illustrated by Figure 4.11. Both halves of the key space are organised the same way: meta-molecules are ordered by 1) type, 2) value within the type. Organising the layer in this manner lets nodes search exactly for the reactants they need for completing reactions. They do so by performing range queries such as supported by works such as [26] and [89] on the second layer.

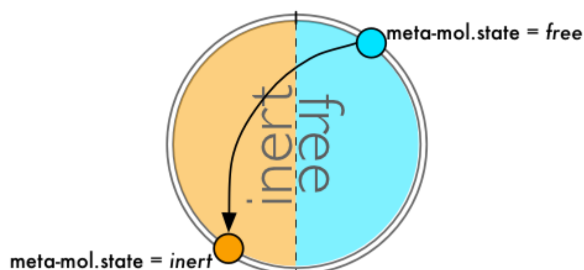


Figure 4.11: Order-preserving layer: as a meta-molecule's state changes, it gets repositioned in the second layer.

As already mentioned, one execution step starts by obtaining a meta-molecule by means of the `random_mol` routine (Algorithm 9, line 2). Although fetching a starting meta-molecule may be a straightforward process when one is able to perform range queries, we still need to somehow avoid all nodes fetching the same meta-molecule at the same time, which could lead to a high probability of conflict during the capture phase. To avoid this problem, some randomness is injected into the process: each node picks a possible free meta-molecule identifier uniformly at random and issues a request for that identifier in the second layer. The receiver of the request, *i.e.,* the node responsible for the range containing the identifier checks its meta-molecules and returns the one with the closest identifier. 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 receiver's responsibility area. It then generates one random identifier for each range and sends two new such requests into the second layer. This process continues until either a meta-molecule has been found or until the whole half key space has been searched with no result. In the latter case, a negative response message is sent to the original requester, after which the termination phase is triggered.

When the requester obtained a first meta-molecule, say mm_1, it needs to find another one, say mm_2 so as to create a pair of reactants. In other words, in the rule of the form **replace** a, b **by** $F(a, b)$ **if** $C(a, b)$, mm_1 and mm_2 will be tested in the two possible orders against C. Note that the request to find mm_2 can be built wisely. Say the condition is $(a + b < 0)$ and that $mm_1 = 17$. Clearly, in this particular case, we need to find some integer lower than -17. This kind of pre-processing allows to drastically reduce the range of possible values. While this reduction is not always possible, it can bring drastic improvement on the complexity of finding the candidates. Once the range has been established, the request is sent to the second layer and processed as explained before, with one significant difference: both free and inert molecules are looked up for the given range. This can result into several candidates being sent to the requester, who may have to choose among them, the rest getting discarded.

Extending this to multiple molecules is conceptually straightforward: if more than two molecules are needed, the process is repeated the relevant number of times. Extending this to multiple rules is again a matter of repetition: once a random molecule is found, the process is repeated for each rule until either a suitable set of reactants is found, or all rules have been tested in which case the molecule is declared as inert and moved into the inert part of the space.

The third step consists in capturing the reactants found. This part is achieved through the protocol presented in detail in Chapter 3.

4.3.3 Higher-order

Higher-order programs contain rules which manipulate other rules as any other molecule in the solution: *rule molecules* can be consumed as well as produced in reactions. Still, there exists differences between rule molecules and data ones. Namely, the former are *n-shot* while the latter are *one-shot*. Once a regular molecule is captured and consumed, it disappears from the multiset. On the other hand, when a rule is being

executed, its molecule has to be present in the solution but it is not consumed in the process, so it can be applied again, possibly concurrently, by other nodes. Still, as rules can be consumed or produced at execution time, they can appear and disappear from the multiset. Recall that they have to be defined statically by the programmer, prior to execution. Still, they may or may not be present in the solution when a node tries to execute it. Therefore, the node has to check whether the rule it is about to execute exists or not. This subsection presents the extensions incorporated into the general algorithm presented in Section 4.3.1 to allow it to support higher-order programs. We address here two crucial issues: *(i)* the discovery of present rules and their execution; and *(ii)* dealing with inconsistencies arising when rule molecules are removed from the multiset.

Tracking rules' existence. Every node periodically tracks the existence of the rules throughout the execution. When it first receives the set of rule molecules, a node tags them as *non-existent*, and assigns them a *recheck time*, *i.e.,* a timestamp stating when it should recheck whether the rule's molecule has appeared in the solution. At the beginning of each execution cycle, before searching for a random meta-molecule, a node goes through this list and singles out rules whose recheck time expired. For each of them, the node sends an *ALIVE* message to the holder of the rule's molecule. If the response is negative, a new recheck time is calculated, and the rule is left in the *non-existent* list. In case the holder replies with a positive message, the rule is transferred to the *existent* list, which keeps track of rules the molecules of which are present in the solution. The execution cycle then resumes with the selection of a random meta-molecule, as explained previously. When a rule becomes *existent*, one needs to be sure that molecules previously declared *inert* cannot become a primary reactant due to the apparition of this new rule. So, if the node is a holder of *inert* meta-molecules, it checks whether it stores meta-molecules which could satisfy the pattern of the newly *existent* rule. Those meta-molecules are then redeclared as *free*, *i.e.,* their state is changed and they are stored in the *free* half of the order-preserving layer again, in this way allowing all of the potential reactions with the newly-existent rule to be performed.

Confirming a rule's existence. Once a rule to execute has been chosen and some molecules have been grabbed with the objective to apply this rule on them, the rule is rechecked for existence, as it might have been consumed by another rule while the molecules were grabbed. Thus, before actually performing the reaction the node inquires the holder of the rule's molecule's presence in the solution. The reaction is carried out if the response is positive. Otherwise, the rule is placed in the *non-existent* list and assigned a new recheck time, while the captured molecules are returned to their respective holders.

Treating higher-order rules. Rule molecules are handled a bit differently than regular data ones, since they represent the functional part of the program. The time diagram of the execution of a higher-order rule is shown in Figure 4.12. Here, $c(R1)$ denotes the node about to execute the rule $R2$, which consumes $R1$. $h(R1)$ represents the node holding $R1$'s molecule, while $h(R2)$ is the node holding the molecule of $R2$. First, $c(R1)$ captures the molecule of $R1$ and then checks for the existence of $R2$ by sending an *ALIVE R2* message to $h(R2)$. After it has received the confirmation, $c(R1)$ is sure that a reaction is going to be performed. Thus, it informs the holder of $R1$ that a reaction is about to be performed with a *DONE* message, and then carries out the reaction.

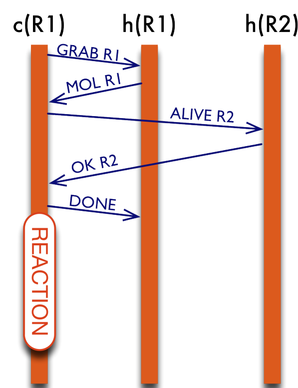


Figure 4.12: Time diagram for higher-order rules.

This last step is done only when a rule molecule is consumed in a reaction. It is needed in order to confirm to the holder that the rule molecule does not exist any more. In between the time it gives the rule molecule to a node and the time it receives the *DONE* message, a holder continues to reply positively to *ALIVE* requests, since it cannot be sure whether the reaction consuming the rule molecule it has given away has been performed or

not; the mere fact of capturing a molecule does not assure a reaction will take place. As we discuss just below, this step may introduce inconsistencies during execution. Still, it is mandatory in order to guarantee the proper detection of inertia.

Potential inconsistencies. As extensively studied in Chapter 3, the chemical programming model enforces mutual exclusion on molecules — a molecule cannot be used in more than one reaction. Consequently, rule molecules, as others, will not be consumed several times. Still, because the flat reactor is decentralised and asynchronous, we need to more carefully look at another potential consistency problem: the case where a rule tries to apply a rule at the same time another node tries to consume its corresponding molecule. In fact, this case can lead to what can be seen as an inconsistency: a node applies a rule after it was consumed by another node. In fact, the protocol lets this happen potentially. We argue that this is not a problem, in the sense that the chemical model is not naturally armed for such a case: According to the chemical model execution model, two active rules at the same depth in the solution can be applied in any order. Again, recall that the only requirement of the model in terms of scheduling is that if at least one rule is active, at least one reaction is performed. Parallelism, and thus non-determinism is the norm. Let assume a program with two rules: hr, a higher-order rule consumes the basic rule r. According to the model, if at some point of the execution, both rules r and hr can be applied, the two possible orders can arise. In fact, once hr is active, it can be delayed arbitrarily. Consequently, if r continues to be applied for a while once hr was applied, the resulting multiset is amongst the set of valid multisets resulting from the possible executions of the program. You can find a more detailed discussion about the inconsistency issue and inertia detection when the higher-order is enabled in the PhD dissertation of Marko Obrovac [77].

4.3.4 Prototype

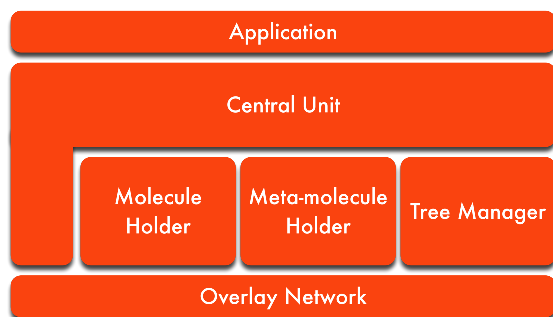


Figure 4.13: Logical view of the entities forming the flat reactor prototype.

In order to better capture the performance of the flat reactor devised, we developed a prototype whose architecture is illustrated in Figure 4.13. The logical architecture of the flat reactor prototype is somewhat similar to that of the hierarchical reactor prototype: The central unit acts as an intermediate module between the application and the functional parts (holders and manager described previously), themselves relying on the overlay network for their communications. Each entity in the picture communicates with the ones directly above and below it. The application transfers the program to the central unit, which then scatters the molecules and multicasts the rule definitions *via* the overlay network. During this period, as for the hierarchical reactor, the tree manager monitors the overlay and builds the node's local state — parent/child relationships with other nodes —, used for multicast requests. More specifically to the flat reactor, the central unit is also in charge of executing the main loop (Algorithm 9). For obtaining random meta-molecules and performing range queries, it consults the meta-molecule holder, which communicates with meta-molecule holders on other nodes in order to fulfil the request. Requests concerning molecules, in particular testing molecules against rules are done by the molecule holder entity. This prototype was developed in Java[3].

[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`.

4.3.5 Experimental evaluation

Test programs

The prototype was first tested on two categories of programs. The first one is *massively parallel*: the same operation is applied to data in parallel without synchronization constraints. This category of programs is represented by the $getMax$ program. Recall that the potential of reactions decreases over time during execution. Note finally that the number of reactions performed , from one execution to the other, is always the same, provided the initial number of molecules in the multiset is constant. The second category of programs is *producer/consumer* programs. This second category exhibit stronger needs in terms of synchronization: once some data has been produced, it is sent over to the consumer. Conversely, the consumer needs to wait for the producer to emit data to start. It can be seen as a composition of massively parallel sub-programs. This second category was embodied by `StringManip` — a program comprising 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 composed of the first λ characters of the original molecule's string, the other containing its remainder. The second rule, *ConcatStr*, takes two molecules as inputs and outputs one which is their concatenation. Thus, *SplitStr* produces the molecules which are going to be consumed by *ConcatStr*. The two rules are circularly dependant. 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.

To assess the proposed platform's responsiveness to changes in the program's set of rules induced by their dynamic addition and removal, we used a simple program which first calculates the sum of some integers and then switches its computation to string concatenation based on integers currently present in the solution. The initial solution contains only integer molecules along with the *sum* rule, which takes two integers and produces their sum. Then, at a random point in time, a new rule molecule appears — that of *switcher*, a rule erasing *sum*'s rule molecule — effectively ending the *sum epoch*. *switcher* injects two rule molecules: *int_to_str*, which turns integers into their string representations; and *str_concat*, which concatenates two string molecules.

Results

The experiments were also conducted on Grid'5000[4] [27]. For each run, the nodes were again spread randomly over the nine geographically-distant sites. Firstly, we evaluated the viability of the platform by executing the first two programs while varying the number of nodes participating in the execution. Figures 4.14 and 4.15 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. This is to be expected since, the more nodes are active, the more reactions can be done in parallel. This is of course especially true when there is a limited number of conflicts at capture time, the nodes using the optimistic protocol described in Chapter 3. The speed-ups obtained are significant. Still, the speed-up obtained for *getmax* is greater than that for `StringManip`. This is due to the difference of the programs' characteristics. The execution time of `StringManip` depends on the sequentiality of events: certain reactions cannot be carried out before others are. In contrast, the execution of *getmax* is highly parallel, as the maximum possible number of reactions can be performed at any given point in time. Also, the execution takes more time to complete for 1000 nodes than for 750 when executing `StringManip`. This is again due to the program's sequentiality: more nodes are in conflict over a subset of molecules since not all available molecules can be used straight away.

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

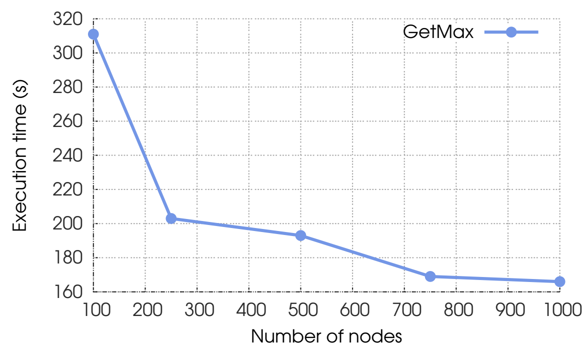


Figure 4.14: Execution time of *getmax* containing 50, 000 molecules.

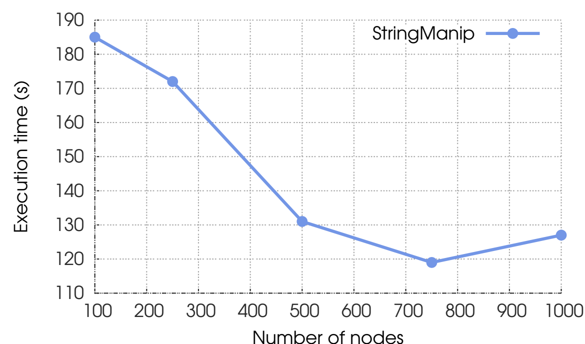


Figure 4.15: Execution time of `StringManip` with 20, 000 molecules.

During the previous experiments, we also monitored the network traffic generated. Our findings concerning the number of messages is that it increases linearly with the number of nodes (for both programs, with a constant number of molecules). Still, the number of messages per nodes decline with the growth of the network, which shows the scalability of the platform on the network side. Finally, we conducted some experiments varying the initial number of molecules while keeping the number of nodes constant (using the *getmax* program). These experiments show that both the execution time and number of messages also increase linearly with the number of molecules. Again, the average number of messages per reaction decrease with the number of molecules. This is due to the fact that there are less conflicts, confirming experimentally the relevance of the adaptive capture protocol, and reinforcing the evidences of the scalability of the platform. The details of these results can be found in [78].

The second part of our experimental campaign dealt with the evaluation of the higher order, under two dimensions: the overhead it brings about to the execution, and its responsiveness, *i.e.,* its quickness to actually stop applying a rule whose molecule has been consumed. About the first dimension, our experimental findings are that the rule-check procedure introduces an overhead which increases with the number of molecules, as it is repeated each time the application of a rule is attempted. Still, the general behaviour of both the execution time and generated network traffic remains linear in the size of the problem.

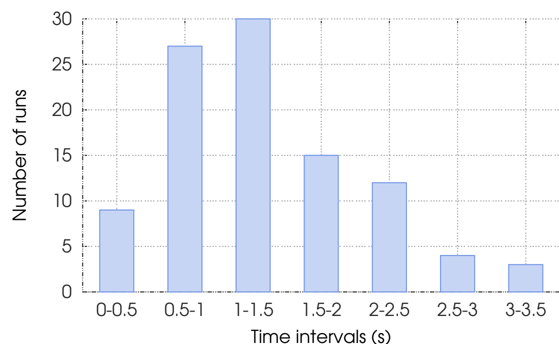


Figure 4.16: Responsiveness of the platform with regards to the time it takes to detect the disappearance of a rule.

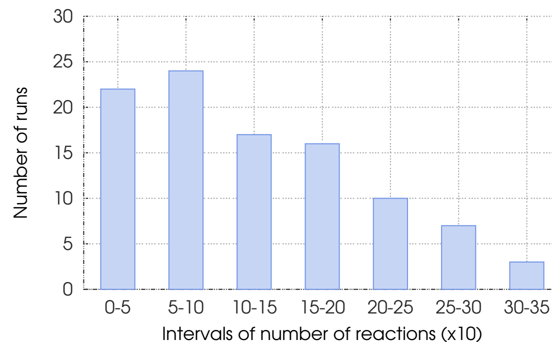


Figure 4.17: Responsiveness of the platform in terms of number of reactions done with a non-existing rule.

Our last experiment, whose results are represented in Figures 4.16 and 4.17, we used the higher-order program described above in order to assess the platform's responsiveness to dynamic changes in the program's execution flow. We executed the program 100 times on a network of 250 nodes, changing the network

configuration each time by randomly choosing the site, the cluster and the port number of each node. We measured the time it took to all of the nodes to detect that the *sum* rule's molecule is not present in the solution any more. At the beginning of each run all of the nodes' internal clocks were synchronised in order to obtain precise measurements. Figure 4.16 shows the number of runs in which the nodes stopped executing the rule after its disappearance from the solution, divided in half-second intervals. Likewise, Figure 4.17 shows the number of reactions carried out during that period.

As explained in Section 4.3.3, this delay manifests itself because *sum*'s molecule's holder replies positively to rule check requests until it has received the confirmation of its consumption. This delay depends on the underlying network's delay. Based on Figure 4.16, we observe that in 81% of runs, nodes stop executing the rule at most two seconds after its disappearance, while the highest delay is 3.5 seconds, observed in only 3% of runs. Even though this might seem as a long period of time, the two-second delay represents only 1% of the overall execution time. In the same vein, Figure 4.17 suggests that in 79% of runs, nodes do at most 200 reactions during the delay, which represents 1% of the total number of reactions done during the execution and less than one reaction per node.

4.4 Conclusion

This chapter discussed our works around the design, development and experimental validation of two distributed runtime systems for chemical programs. The first, *hierarchical* reactor is conceptually simple and builds an execution tree in which the inert sub-solutions are fused until reaching the root of the tree. The hierarchical reactor does not need complex algorithmic, just a bit of care when parent nodes receive inert sub-solutions from their children, to avoid losing the work achieved in their subtree. The hierarchical reactor suffers from its shape, potentially leading to great disparities in terms of workload across nodes. It is also hard to extend to support higher-order programs. In contrast, the *flat reactor* is fully decentralised and does not include any by-design non-uniform load balancing. It includes a molecule discovery protocol based on a two-layer DHT and complex queries allowing to detect early that some molecules may not react anymore. This pruning scheme was not fully exploited in our work, as anyway all molecules are scanned at least once by some node. Extending such a complex querying-based inertia detection could allow to significantly reduce execution time when a lot of molecules cannot react anymore. Due to the full decentralisation and the subsequent need for a molecule discovery mechanism, a capture protocol is needed to avoid several nodes discovering overlapping sets of molecules to *allocate* them at the same time. The flat reactor leverages the adaptive capture protocol of Chapter 3 to address this issue. Finally, the flat reactor supports the higher-order.

Both reactors have been implemented, deployed and evaluated over a geographically dispersed platform. To our knowledge, these works were the first to distribute the execution of chemical programs at such a scale, giving hints for the potential usage of the paradigm over massively distributed platforms.

This chapter closes the part of this dissertation dedicated to the model itself and its implementation in distributed settings. Now that some barriers towards its practical usage have been lifted, it is time to see in more detail how the programming model it offers can be used to help programming autonomic systems, and more specifically how it can be used to specify service composition and workflows with adaptiveness capabilities.

Chapter 5

Decentralising Workflow Execution

Previous chapters dealt with the chemical programming model and its implementation in distributed settings. The two following chapters are more about how it can be used to program (workflow-based) applications and help bring decentralisation and adaptiveness to their execution. This chapter deals with the decentralisation of workflow execution. It shows how a set of agents can coordinate themselves to share the coordination of the execution. To this end, we devise a shared-space architecture on top of which chemically-inspired abstractions are used to express the workflow and its enactment. The implementation of these concepts into the GinFlow middleware is detailed and their experimental validation discussed.

Joint work with:

- ⋆ Héctor Fernández (PhD student, Inria)
- ⋆ Thierry Priol (Senior researcher, Inria)
- ⋆ Matthieu Simonin (Research engineer, Inria)

5.1 Workflows, workflow managers, and decentralisation

As we have discussed more extensively in Introduction, service composition is an attractive paradigm to build complex applications out of more basic yet well-defined basic services. Service compositions (or *composite services* are most commonly *temporal* composition of such building block services. Industrial organization such as the OASIS consortium proposed many standards to help program and manage service-oriented architectures. Architectures following these standards (most of them following an XML-based grammar) have been notoriously known as *web services*, a term that is now commonly used to describe any flavour of web-based application, would it be built over SOAP or using a RESTful approach. BPEL (Business Process Execution Language) is one of these standards, specifically targeted at describing service composition. It can be seen as the assembly language for service composition, embedding a lot of implementation details (protocols used, service locations, . . .) and being globally very verbose. The abstraction level provided by BPEL does not allow the programmer to concentrate on the logic of the composition properly.

This lack of a high-level programming model for service composition is in particular a barrier in the scientific world. In many scientific domains, simulation experiments is now the dominant paradigm for research. These compute-intensive simulations commonly compose functional building blocks together. Such compositions are referred to as (scientific) *workflows*. The concept of workflow is very similar to service composition. More formally, a workflow can be represented by a graph in which each vertex is a function, or *service* and each edge a data dependency or a control dependency between services. Workflows are now widely used, reused and shared among scientific communities, as for instance through the myExperiment[1] workflow sharing platform.

[1] `http://www.myexperiment.org`

Workflow management systems such as Taverna [103], Kepler [69] and MOTEUR [53] have focused on providing tools to design and execute scientific workflows with a high level of abstraction. These workflow managers all rely at some point on a centralised orchestrator responsible for coordinating the workflow execution. The drawbacks of such an element have been discussed in Introduction. While limited, some workflow research focused on enabling direct cooperation between services in order to allow executions even in the case no central engine is available. We will come back to these works later in the chapter, for the sake of comparing our proposal with the state of the art. Getting rid of a centralised manager and allows for horizontal scalability, and raises the level of reliability and even privacy. It globally moves the system towards more autonomy. Please refer back to Chapter for this discussion.

In Section 5.2, we present first an architecture aiming at enacting workflows in a decentralised fashion. This architecture relies on a shared space for coordination between services involved in the workflow. We then present in Section 5.3, based on this architecture, a programmatic way to specify workflows over decentralised settings. The implementation of these concepts was done through the development of a research prototype during Héctor Fernández' PhD [44]. This software prototype was later turned into the more mature GinFlow workflow manager[2] presented in Section 5.4. In particular, GinFlow's software architecture and resilience mechanisms are detailed. GinFlow's scalability and resilience, and thus the viability of these concepts are validated through a set of experiments conducted over the Grid'5000 platform in Section 5.5. Before concluding the chapter, Section 5.6 positions this work in the landscape of distributed workflow coordination research.

5.2 Shared-space based coordination

The execution model we have proposed is to enact a decentralised execution through the splitting of the traditional workflow engine (or composition orchestrator) into a set of agents co-responsible for the execution of the workflows. The architecture includes two types of components, namely i) the *shared space* which contains the description of the workflow and ii) the *service agents* (SAs) which will share the coordination work in executing the workflow described in the shared space. During enactment, each time the execution moves forward after some action performed by some agent, the shared description is updated so as to reflect the execution progress. The SAs are essentially workers that encapsulate the invocation of the services. This encapsulation includes an engine able to read, interpret and update the information contained in the shared space. For instance, when a SA completes the invocation of a service and collects the result, it pushes this information to the shared space, allowing another service agent, which was waiting for this result, to collect it and use it as input to invoke the service it encapsulates.

A possible scenario of execution in such an architecture is depicted in Figure 5.1: i) the shared space (represented as a circle surrounding the graph) is fed with the description of the workflow and each SA collects from it the information it needs to actually takes its part in the coordination (incoming and outgoing dependencies, and services to call); ii) The SA checks the information received and the ones that can start doing something (*i.e.,* those responsible for services with no incoming dependencies) actually trigger the execution of the workflow; iii) Once a SA completes the execution of its services, it sends its output to its dependencies and updates the shared space so as to reflect that the workflow moves forward which make the subsequent parts of the workflow able to start in their turn; iv-v-vi) this process is repeated until the whole execution is completed.

The relation between such an architecture and HOCL is kind of straightforward: replace *shared space* by *multiset* and *Service Agents* by HOCL engines applying HOCL rules on part of this multiset and you are done. Also, having a declarative programming style helps to easily express the coordination. To sum up, in the context of this decentralised workflow executor, the initial multiset will contain the description of the workflow, and the rules will express how to modify this description so as to reflect the execution is

[2] `https://ginflow.inria.fr`

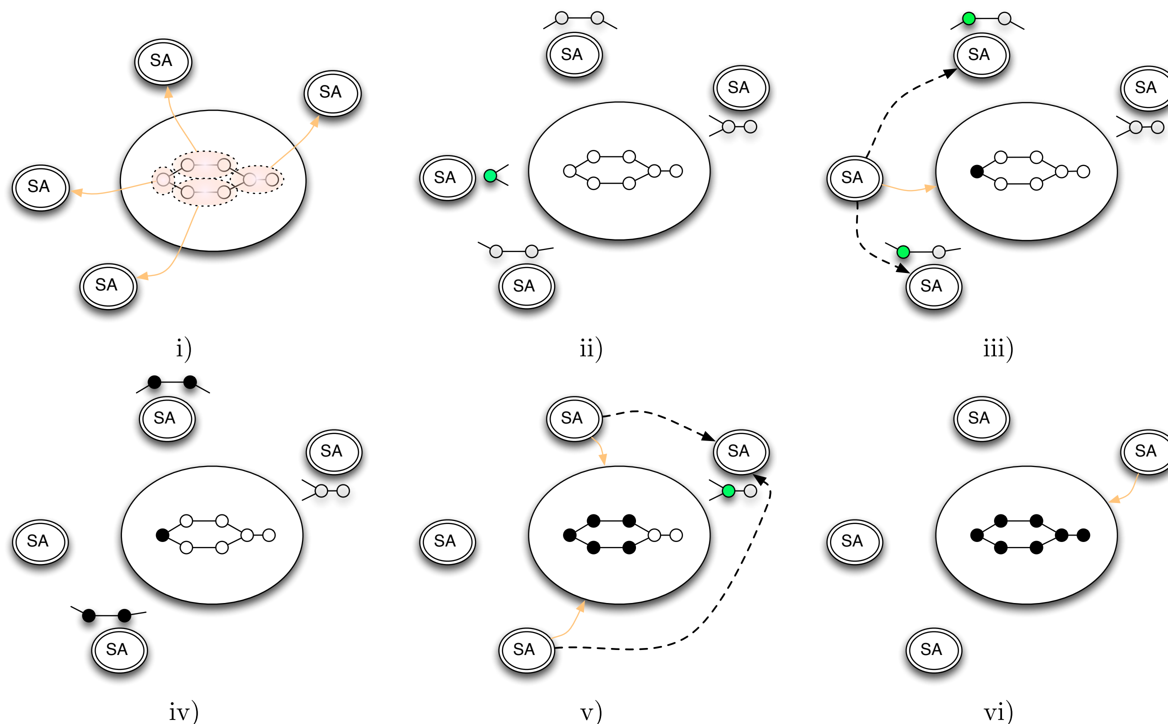


Figure 5.1: An example of shared-space coordination.

moving forward. As we will see in details, the needs are simple: i) expressing that a service is called (once all its dependencies have been satisfied) and ii) expressing that data are moved once a service's call has been completed.

5.3 Chemical workflow specification

We now use an extended version of HOCL named HOCL$_{flow}$. It is introduced specifically for workflow expression. HOCL$_{flow}$ provides extra syntactic facilities over HOCL. Firstly, it includes lists (which are not natively supported by HOCL). Secondly, it offers some syntactic sugar avoiding to double mention *catalysts* in reaction rules. More precisely, rules of the form: **replace-one** X **by** X, M where X and M are multisets of molecules, can be written: **with** X **inject** M. Thirdly, it includes reserved keywords for specific atoms to ease the workflow management. They will be explained when used hereafter.

A workflow is mostly a set of tasks to be executed in a certain order. A task can be seen as an abstract function, to be implemented by a service. A workflow can be represented as a directed acyclic graph (DAG), as illustrated in Figure 5.2.

An HOCL workflow definition is composed of as many subsolutions as there are tasks in the workflow. The HOCL code for the workflow in Figure 5.2 is given in Figure 5.3. Each subsolution is similar and contains (initially) four atoms. The first two tuple atoms, prefixed by the reserved keywords SRC and DST, specifies the incoming and outgoing dependencies of the task, respectively. The two latter atoms provide the data needed to invoke the service, namely, its name (in the SRV atom) and its parameters (in the IN atom). Recall that SRC, DST, SRV and IN are reserved keywords in HOCL$_{flow}$.

Still, if the code in Figure 5.3 is given as input to an HOCL interpreter, it will not trigger anything—it does not comprise rules. Thus, we need to design a set of rules able to, when combined with a workflow description, execute the specified workflow. We now devise a set of three simple rules which constitutes the minimal set of rules allowing any workflow to run. We call these rules *generic* and prefix their name with

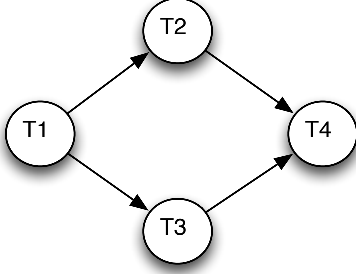


Figure 5.2: A simple workflow DAG.

3.01 $\langle$
3.02 $\quad T_1 : \langle \text{SRC} : \langle\rangle, \text{DST} : \langle T_2, T_3\rangle, \text{SRV} : \texttt{s1}, \text{IN} : \langle \texttt{input}\rangle\rangle,$
3.03 $\quad T_2 : \langle \text{SRC} : \langle T_1\rangle, \text{DST} : \langle T_4\rangle, \text{SRV} : \texttt{s2}, \text{IN} : \langle\rangle\rangle,$
3.04 $\quad T_3 : \langle \text{SRC} : \langle T_1\rangle, \text{DST} : \langle T_4\rangle, \text{SRV} : \texttt{s3}, \text{IN} : \langle\rangle\rangle,$
3.05 $\quad T_4 : \langle \text{SRC} : \langle T_2, T_3\rangle, \text{DST} : \langle\rangle, \text{SRV} : \texttt{s4}, \text{IN} : \langle\rangle\rangle$
3.06 $\rangle$

Figure 5.3: HOCL definition of a simple workflow.

4.01 gw_setup =
4.02 **replace-one** $\text{SRC} : \langle\rangle, \text{IN} : \langle\omega\rangle$
4.03 **by** $\text{SRC} : \langle\rangle, \text{PAR} : \text{list}(\omega)$

4.04 gw_call =
4.05 **replace-one** $\text{SRC} : \langle\rangle, \text{SRV} : \texttt{s}, \text{PAR} : \ell_{\text{PAR}}$
4.06 **by** $\text{SRC} : \langle\rangle, \text{SRV} : \texttt{s}, \text{RES} : \langle \texttt{invoke}(\texttt{s}, \ell_{\text{PAR}})\rangle$

4.07 gw_pass =
4.08 **replace** $T_i : \langle \text{RES} : \langle\omega_{\text{RES}}\rangle, \text{DST} : \langle T_j, \omega_{\text{DST}}\rangle, \omega_i\rangle,$
4.09 $T_j : \langle \text{SRC} : \langle T_i, \omega_{\text{SRC}}\rangle, \text{IN} : \langle\omega_{\text{IN}}\rangle, \omega_j\rangle$
4.10 **by** $T_i : \langle \text{RES} : \langle\omega_{\text{RES}}\rangle, \text{DST} : \langle\omega_{\text{DST}}\rangle, \omega_i\rangle,$
4.11 $T_j : \langle \text{SRC} : \langle\omega_{\text{SRC}}\rangle, \text{IN} : \langle\omega_{\text{RES}}, \omega_{\text{IN}}\rangle, \omega_j\rangle$

Figure 5.4: Generic workflow enactment rules.

gw (for *generic workflow*). They are defined in Figure 5.4.

The first two rules prepare and call the service implementing a task. They will act inside the subsolution of a service. The third one acts at the workflow level (and will consequently appear at the outermost multiset). It moves data between services as specified by the dependencies. Rule gw_setup detects that all the dependencies of a task have been satisfied by checking that the SRC : $\langle\rangle$ atom is empty (it does not have to wait for another input anymore) and fills the parameter list in the PAR atom.[3] This activates the gw_call rule which calls the service (with the list of parameters in the PAR atom), collects the result, and puts it back in the task's subsolution (in the RES) atom. Rule gw_pass is responsible for transferring results from one source to one destination. Its scope spans two services (one source and one destination). It is triggered after the result has been obtained and placed in the RES atom of the source. It moves the resulting value from the source to each declared destination, through its repeated application. It updates SRC : $\langle\rangle$ and DST : $\langle\rangle$ each time it is applied, to remove a satisfied dependency.

Adding these rules (gw_setup and gw_call within each subsolution, and gw_pass inside the global solution) to the abstract workflow described in Figure 5.3 makes it a fully functional workflow execution program, to be interpreted by an HOCL interpreter. The DAG is the only requirement from the user because the generic rules can be inserted automatically prior to execution.

The HOCL workflow description is internal to GinFlow and is not required directly from the user, who can provide a more user-friendly description of the workflow, for instance using a JSON format (as we detail

[3]Prior to execution, the IN atom can contain data. This input, combined with the data received from other tasks will constitute the list of parameters of the service (stored in the PAR atom). The *list*() function creates a list.

later in Section 5.4). This JSON definition will be then translated internally so as to obtain the required HOCL representation.

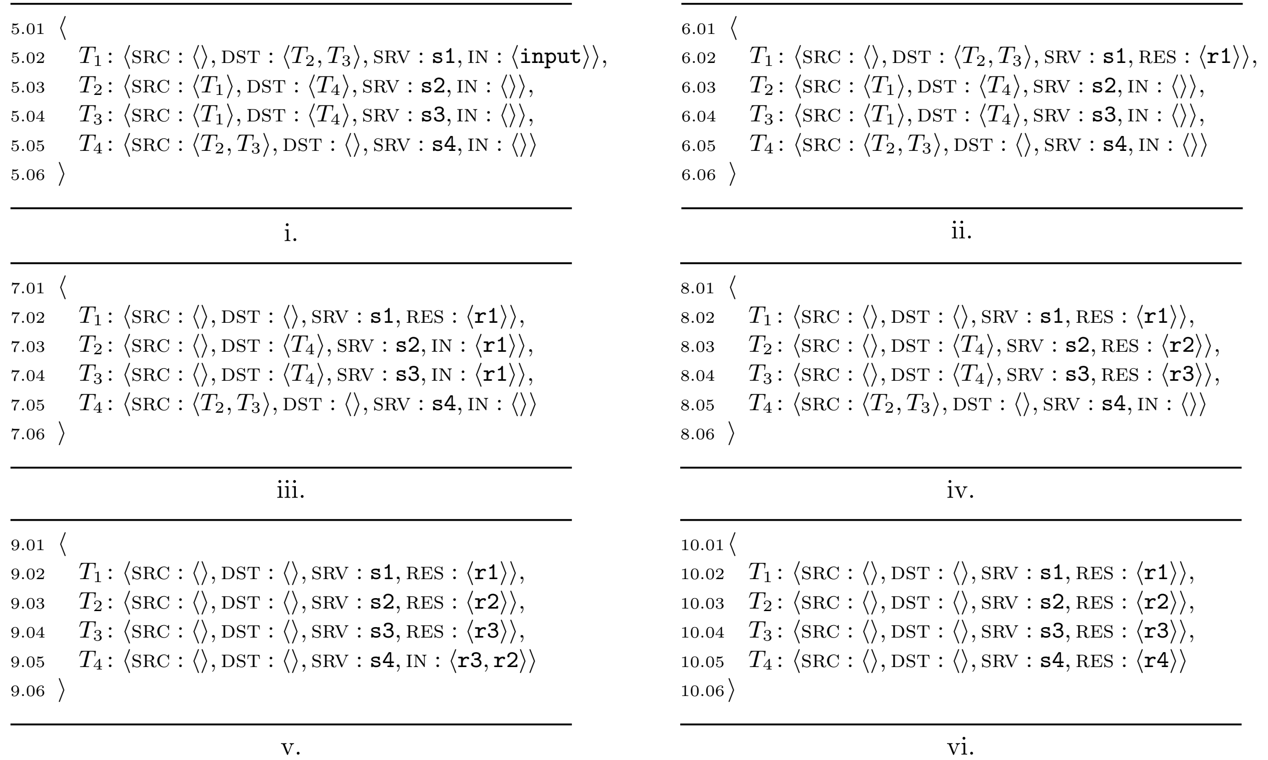


Figure 5.5: Simple workflow's execution scenario.

The evolution of the multiset during execution is illustrated in Figure 5.5. Between state i. and state ii., the agent responsible for T_1 applied both gw_setup and gw_call and the result of `s1` appears in a newly created RES : ⟨⟩ atom. Then, to reach State iii., the gw_pass rule was applied twice, once for each needed transfer towards T_2 and T_3's subsolutions. Then, in parallel, respective service agents responsible for T_2 and T_3 was able to gw_setup and gw_call in their turn so as to create their respective results `r2` and `r3`. Towards state v., and still in parallel, the same SAs apply gw_pass so as to move their results to T_4. The arrival order of results into T_4's subsolution is not predetermined. The multiset reaches its final state (vi.) with the SA responsible for T_4 executing `s4` and getting its outcome.

5.4 GinFlow

This section presents the GinFlow workflow middleware. It is the result of a significant refactoring and extension of the early prototype developed by Héctor Fernández and presented in [46]. It was made a more mature, usable and extensible software by Matthieu Simonin, research engineer at Inria. Its code was released under the LGPL-3 license in 2016. It is registered at the APP (*Agence de Protection des Programmes*) under the number IDDN.FR.001.46018.000.S.P.2015.000.10600.[4]

GinFlow sits on the HOCL language and interpreter which consists in 18 000 lines of Java code developed by Yann Radenac, during his PhD defended in 2007 [83]. The middleware itself represents approximately

[4] To download and try it, please go to `https://ginflow.inria.fr`.

2 000 more lines of Java code. Figure 5.6 depicts the general GinFlow architecture. This architecture is made of three layers. In Section 5.4.1, we present the bottom layer of the architecture and highlights the internals of the core engine. In Section 5.4.2, the resilience of this layer is described. Section 5.4.3 focuses on the intermediate level of the architecture, the executor layer in use in GinFlow. Section 5.4.4 deals with the upper level providing ways to users to interact with GinFlow.

5.4.1 The HOCL distributed engine

At the core (the lower layer in Figure 5.6), the GinFow middleware consists in a distributed engine, which implements the concepts of service agents (SAs) communicating through a shared space containing the description of the current status of the workflow. In GinFlow, a SA is composed of three elements. The first element is the service to invoke, any wrapper of an application representing this service, or any interface to the service enabling its invocation. The second element is a storage place for a local copy of the multiset. The local copy acts as a cache of the service's subsolution. The third element is an HOCL interpreter that reads and updates the local copy of the multiset each time it tries to apply one of the rule in the subsolution. A SA stores locally only the status of the portion of the workflow it is responsible for. The cache is only read and written locally by this single interpreter, and often pushed back (written) to the shared space, ensuring no coherency problems can arise.

Once each SA collected data from the multiset, it communicates directly with other agents in a point-to-point fashion, for instance when transmitting data between two workflow tasks. Recall that, as earlier described in Section 5.3, enactment rules have to be injected into the workflow description prior to its execution, (*i.e.,* before any agent starts collecting its initial information from the multiset).

More precisely, the rules presented in Section 5.3 do not enable a decentralised execution by themselves. In particular, looking carefully at the gw_pass rule, whose detailed functioning was until now eluded, we can see it spans several subsolutions and it is thus supposed to act from outside subsolutions since it requires to match the atoms from several subsolutions. In the GinFlow environment, it was modified to act from within

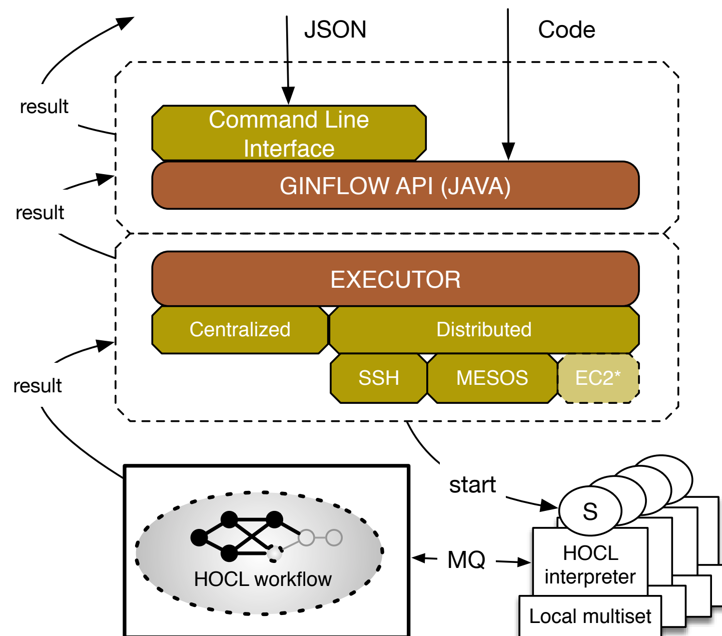


Figure 5.6: Architecture of the GinFlow software.

a subsolution: once the result of the invocation of the service it manages is collected, a SA triggers a local version of the gw_pass rule which calls a function that sends a message directly to the destination SA. It also sends a message to the multiset so as to update the status of the workflow. When the message is received in the multiset, it is simply pushed to the right subsolution.

In GinFlow, the inter-agents communications rely on a message queue middleware which can be either Apache ActiveMQ or Apache Kafka. The choice for one or the other depends on the level of resilience needed by the user, as we detail below.

5.4.2 Resilience

In this section, we present a mechanism that automates the recovery of failed server agents. When one SA fails, for instance following a hardware crash, another SA will be automatically started to replace it. The difficulty here stands in the ability to replay the work done by the faulty SA prior to its failure.

The state of a SA is reflected by the state of its local solution. Changes in the local solution can result from two mutually exclusive actions : i) reception of new molecules and ii) reduction of the local solution through the action of the local HOCL engine it embeds. A reduction phase is systematically triggered when new molecules are received. Note that this reduction modifies the state of the local solution only if some rule were actually applied during reduction. The SA lifecyle is thus a sequence of receptions and reductions. Consequently, if the system is able to keep the information of all molecules received by a SA and resend them in the same order to a newly created SA, this second SA can recover the state of the first one.

This *soft-state* behaviour of the SAs let us envision a fault-recovery mechanism which will rebuild the state of a faulty SA. It is assumed that the actual services invoked are *idempotent*, or at least free from non-desirable side effects since they can be called several times in case the fault-recovery mechanism is triggered. Still, some results may be received several times by some of the successor nodes of a recovered SA, in case the first SA failed after invoking the service but before finishing the transmission of its results to all of its successors. However, the *one-shot* nature of the gw_setup and gw_call rules ensures that the successors will take into account only the first result received and thus prevent a cascade of useless re-executions.

In GinFlow's implementation, this resilience feature leverages Kafka's ability to persist the messages exchanged by the SAs and to replay them on demand to facilitate the implementation of this mechanism. Section 5.5.3 will present an evaluation of GinFlow's resilience.

5.4.3 The executors

The role of the executor is to prepare, deploy the distributed engine, and bootstrap the workflow in a specific environment, either centralised or distributed. The centralised executor relies on a single HOCL interpreter to execute the workflow. A distributed executor will (1) claim resources from the infrastructure and (2) provision the distributed engine (the SAs and the multiset) on them. The deployment is made of three phases to be described hereafter: *preparation*, *deployment*, and *bootstrapping* phases.

Preparation phase. Users only deal with abstract workflows, *i.e.,* specify the workflow itself without needing to deal with enactment considerations. In other words, the user provides some description, using the GinFlow API to be described in Section 5.4.4, that will be translated into its HOCL equivalent code, as illustrated in Figure 5.2. To allow the enactment of the workflow, rules need to be injected to produce a concrete workflow description. This consists in adding the gw_setup, gw_call, and gw_pass rules into the HOCL workflow representation. This injection occurs during the preparation phase and is performed transparently for the user.

Deployment phase. In the current version of GinFlow, two distributed executors are available: SSH-based and Mesos-based. The SSH-based executor starts the SAs on a set of machines specified by the user in the GinFlow configuration file. It deploys the SAs over the machines in a round-robin fashion. The Mesos-based executor delegates the deployment of the SAs to the Mesos scheduler [56]. Besides these executors,

the abstract nature of the code allows GinFlow to be easily extended with other executors (such as an EC2 executor to run GinFlow's distributed engine on an EC2-compatible cloud).

Bootstraping phase. Once the SAs are deployed over the computing nodes, the workflow execution is started by sending its initial description (including the injected rules) to the SAs. These communications rely on the same communication layer as the execution: ActiveMQ or Kafka, as configured by the user. Upon reception, each SA will start reducing its own solution and thus executing the workflow.

5.4.4 Client interfaces

We here describe how the user interacts with GinFlow. The user is given two entry points: GinFlow's command line and GinFlow's Java API. The former consists in defining the workflow in a JSON-formatted file with the specification of all the tasks and the links between them. The latter uses the internal Java API of GinFlow to describe the DAG programmatically. We again use the workflow depicted in Figure 5.2 for the sake of illustration. Assuming the JSON file describing the workflow was saved under the name `wf-1.json`, the command line below starts the workflow using the centralised executor, *i.e.,* using only one HOCL interpreter running in the same JVM as the main program:

```
java -jar hocl-workflow-2.0.0-SNAPSHOT.jar launch -w wf-1.json
```

To distribute the execution over different processes but still on the same physical computing node, one can specify an alternate executor on the command line. For instance, adding `-e ssh` to the command line distributes the execution using the SSH protocol. Without any further configuration except a passwordless SSH connection, this will result in distributing three service agents, each one running an HOCL interpreter locally and communicating through an embedded version of ActiveMQ, the by default broker started by the GinFlow executor prior to workflow execution. To distribute the SA processes over different machines, or to rely upon one's specfific broker, the user can specify a set of options either through the `-o` switch in the command line, or by using a configuration file following the YAML format.

For more advanced usages, GinFlow exposes a Java API. This programming interface gives the programmer the possibility to define easily higher-level workflow patterns (*e.g.,* a sequence of tasks) and to configure the options of the executor and even start the exection, for instance to start GinFlow from another Java program. Listing 1 exposes the Java definition of the workflow.

5.5 Experimental validation

In this section, we present the results of the experimental evaluation of GinFlow. The results exhibited in the following revolves around i) performance and scalability, ii) the impact of choosing different combinations of executors and messaging middleware, and iii) the SA recovery mechanism presented in Section 5.4.2. Note that we considered only distributed settings, as it constitutes the natural playground of GinFlow and our motivation for this work.

Experimental setup. All the experiments were run using up to 50 multi-core computing nodes of the Grid'5000 testbed, connected through 1Gbps Ethernet. Debian/Linux Wheezy was installed on the nodes and we used ActiveMQ 5.6.0 and Kafka 0.8.1.1 as the communication middleware. Mesos 0.20 was also installed on the nodes. A total of 800 cores and 6 TB RAM were used. In the deployments, the number of SAs per core was limited to two, which allowed to deploy up to 2 603 services.

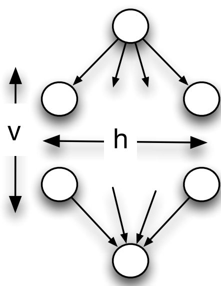
Workflow shapes. Possible shapes of workflows are virtually infinite, however four major patterns, namely *split*, *merge*, *sequence* and *parallel* have been recognised to cover the basic needs of scientific computational pipelines [84]. We consequently decided to conduct the experiments of Sections 5.5.1 and 5.5.2 using a *diamond* shape, depicted in Figure 5.7 and covering the four aforementioned patterns. The shape of diamond workflow varies according to two parameters: h (for *horizontal*), the number of services in parallel at each

```json
{
  "name"    : "wf-1",
  "services": [{
    "name": ["1"],
    "srv" : ["echo"],
    "in"  : ["1"],
    "dst" : ["2","3"]
  }, {
    "name": ["2"],
    "srv" : ["echo"],
    "in"  : ["2"],
    "src" : ["1"],
    "dst" : ["4"]
  }, {
    "name": ["3"],
    "srv" : ["echo"],
    "in"  : ["1"],
    "src" : ["4"],
  }, {
    "name": ["4"],
    "srv" : ["echo"],
    "in"  : ["4"],
    "src" : ["2","3"]
  }]
}
```

```java
Workflow workflow =
  Workflow.newWorkflow("wf")
    .registerService("1", Service.newService("1")
        .setDestination(Destination.newDestination()
           .add("2")
           .add("3"))
        .setCommand(Command.newCommand("echo"))
        .setIn(In.newIn().add("1")))
    .registerService("2", Service.newService("2")
        .setSource(Source.newSource().add("1"))
        .setDestination(Destination.newDestination()
          .add("4"))
        .setCommand(Command.newCommand("echo"))
        .setIn(In.newIn().add("2")))
    .registerService("3", Service.newService("3")
        .setSource(Source.newSource().add("1"))
        .setDestination(Destination.newDestination()
          .add("4"))
        .setCommand(Command.newCommand("echo"))
        .setIn(In.newIn().add("3")))
    .registerService("4", Service.newService("4")
        .setSource(Source.newSource().add("2").add("3"))
        .setCommand(Command.newCommand("echo"))
        .setIn(In.newIn().add("4")));
```

Listing 1: Defining workflow either from JSON (left) or Java (right)

stage, and v (for *vertical*), the number of stages. These workflows come in two *flavours*: *simply-connected* tasks and *fully-connected* tasks. The former is used as base reference and the latter to stress test the engine in terms of messages exchanged across the distributed environment. Since we are mainly interested in the coordination time, the tasks (up to 1 000) themselves simply simulate a dummy program with a near-zero execution time.



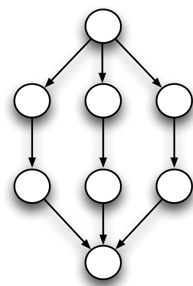
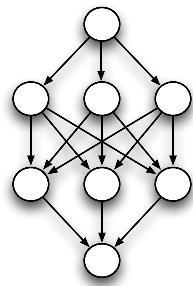



Figure 5.7: Diamond workflows, simple and fully connected.

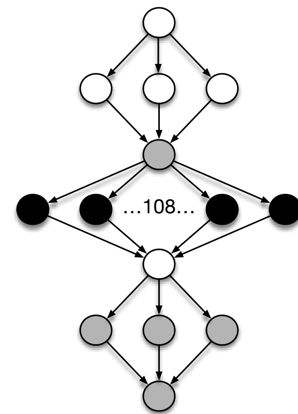


Figure 5.8: The Montage workflow.

Section 5.5.3 uses a workflow built using the Montage toolbox, a set of computational tools building images of the sky out of pictures taken by telescopes. The particular workflow used is one of the classical

Montage workflows. It outputs a 3-degree centered image of the M45 star cluster (often referred to as the *pleiades*) composed of 100 million pixels. It is depicted in Figure 5.8 with the category of duration, in seconds, of its 118 tasks.

5.5.1 Performance

Figure 5.9 shows the timespan for both diamond configurations. It has been obtained by measuring the execution time for all diamond shapes where $width \times height \in [1, 51]^2$ with a step of 5 on each dimension. We observe a sustained time increase when the number of services grows. In a simply-connected configuration, the coordination time of the whole workflow execution is close to 155 seconds for 51×51 services. This time includes the exchange information between linked services and the update of the shared multiset. These results are expected because the number of tasks has a direct influence on the time needed for the reduction of sub-solutions: in an HOCL engine, the complexity of the pattern matching process depends on the size of the solution. However, the load of evaluating molecules in the global solution is distributed among all co-engines preventing a combinatorial explosion. The same behaviour is observed with the fully-connected configuration. In this case, we observe the higher cost of coordination and the impact of connections between services. Since a service has to wait for all services of the previous layer, an implicit coordination barrier is set in each service growing the final execution timespan (up to 530 seconds for 51×51 services). We can also identify this effect on the projection of the vertical services on the time-vertical plane which shows a higher slope than the time-horizontal one. While a chemical engine has a potential performance bottleneck for a large set of services, increasing the number of co-engines limits the final coordination time due to the parallelism, ensuring the scalability of GinFlow in distributed settings.

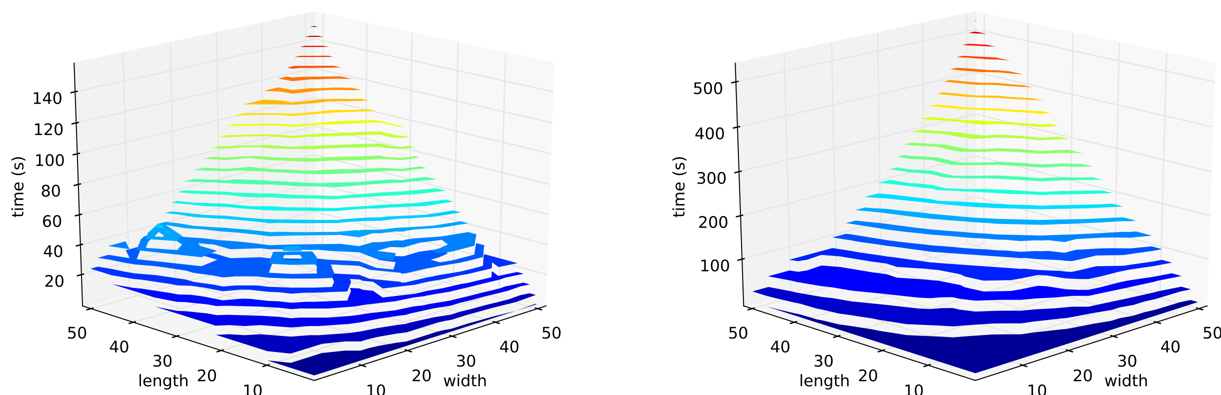


Figure 5.9: Coordination time of diamond workflows, simple-connected (left) and fully-connected (right).

5.5.2 Impact of the executor and messaging middleware

Figure 5.10 depicts the average time (computed over ten runs) to execute of a 10×10 diamond workflow in each executor/communication middleware combination for different number of nodes. It shows the duration of each step: (a) the deployment time (the time taken by the executor to deploy the SAs) and (b) the execution time. Note firstly that the deployment time depends on the scheduling strategy used. The SSH-based executor starts SAs in a round-robin fashion on a preconfigured list of nodes. As the SSH connections are parallelized, the deployment time only slightly increases with the number of nodes. GinFlow, when run on top of Mesos, starts one SA per machine for each offer received from the Mesos scheduler. Thus, increasing the number of nodes will increase the number of machines in each offer and consequently the parallelization

in starting the SAs. This explains the linear decrease of the deployment time observed for the Mesos-based executor.

About the impact of the messaging middleware, ActiveMQ outperforms Kafka as the execution time is approximately 4 times higher with the latter. Nevertheless, the choice of deploying GinFlow with Kafka is justified by the resilience it helps to provide, as it is experimented in the next section.

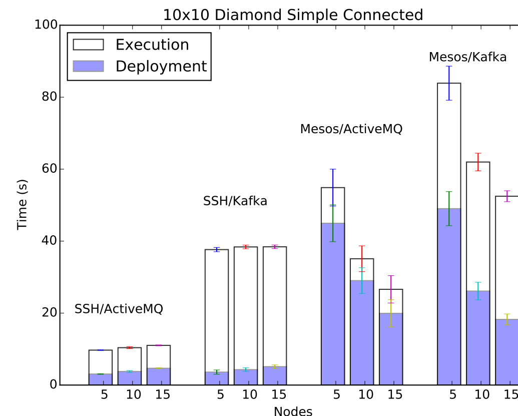


Figure 5.10: Execution time *vs* configurations.

5.5.3 Resilience

We finally evaluated the resilience mechanisms exposed in Section 5.4.2. These experiments were based on the Montage workflow whose shape and duration of tasks are given in Figure 5.8, and made use of the Mesos-based executor (to restart agents) and Kafka as the communication middleware (to persist and replay messages). Note that the services taken from the Montage toolbox are idempotent, in the sense that a service can get restarted safely in the case of a failed previous attempt of executing this service (in the same workflow). The methodology for injecting failures was the following: each running agent failed with a predefined probability p, but never before a certain amount of time T elapsed since they started executing their task. Note that a restarted agent can fail again. Thus, we can expect $\frac{p}{1-p} \times N_T$ (5.1) failures where N_T is the number of services whose duration is greater than T.

For each chosen value of p and T, we repeated the execution of the workflow up to 10 times and averaged their resulting values. The bar plots of Figure 5.11 show the execution time with different values for p and T. The dashed line common to every subplot corresponds to the execution time without any failure. In this case the execution time is 484s in average with a standard deviation of 13.5 seconds.

Firstly, when $T = 0$, increasing p allows to appreciate the recovery velocity. We observed 26, 114 and 487 failures in average for $p = 0.2$, $p = 0.5$ and $p = 0.8$ respectively. These observations conform to the results obtained with Formula 5.1 with $N_0 = 118$. Figure 5.11 shows an increase of $3s$, $36s$ and $208s$ respectively in the average execution time of the workflow, showing that the recovery velocity slightly decreases when increasing the failures rate. In other words, the ratio between the overhead and the number of failures slightly increases.

Secondly, choosing $T = 15s$ experimentally confirms that services can recover even if they fail after some processing. Moreover, note that 95% of the services of this workflow have a running time greater than $15s$ and thus may be affected by failures. Still, the durations of the services in the large parallel part of the workflow are quite heterogeneous: from $60s$, and up to $310s$. Consequently, some services in this part can recover without impacting the global execution time, as their total execution time will still be under the execution time of that of other services running free of failure. In contrast, the execution time of the workflow is more sensitive to the failures of some specific services such as the intermediate merging ones. These two considerations can explain the increase of the standard deviation observed in Figure 5.11.

Finally, choosing $T = 100s$ impacts exclusively services in the large parallel part of the workflow. While the observed increase in the execution time necessarily becomes more important (but still in accordance with Formula 5.1, it illustrates the ability of GinFlow to enact real-world workflows reliably over very unstable infrastructures.

5.6 Decentralising workflow execution in literature

The idea of describing direct interactions between services in the execution of a composition was proposed in the concept of *choreography of web services* [13]. Choreography allows to describe at once all the interactions

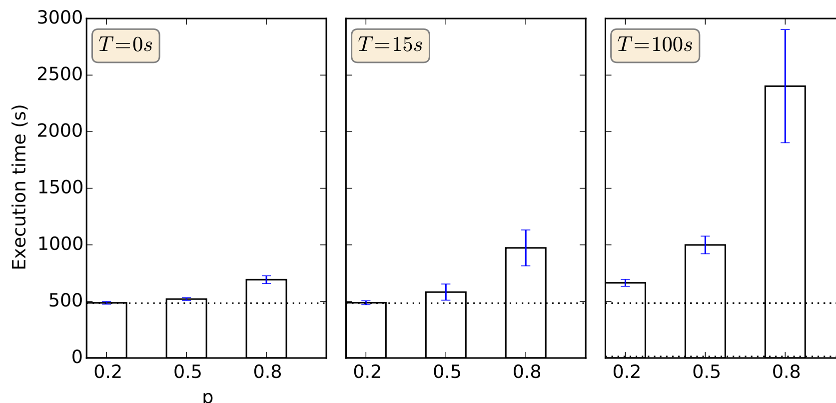


Figure 5.11: Execution time with different failure scenarios.

that will take place at execution time. It allows to be more precise than orchestration in which the execution is seen from a single point of view. As detailed in [44], the choreography models proposed in literature is quite static and increase the tight coupling between participants. Also, the concept remained quite abstract over the years. Finally, note that adapting a choreography is difficult issue, subject to be tackled, but not in the context of choreography but in the context of GinFlow, in the next chapter.

Decentralising the coordination of the workflow execution was first pursued relying on direct interactions (based on messaging) between services. In [21], the authors introduce *service invocation triggers*, a lightweight infrastructure that routes messages directly from a producer service to a consumer one, where each service invocation trigger corresponds to the invocation of a service. In [73], a distributed execution engine is proposed based on a peer-to-peer architecture wherein nodes (similar to local engines) are distributed across multiple computer systems, and communicate by direct point-to-point notifications. A continuation-passing style, where information on the remainder of the execution is carried in messages, has been proposed in [104]. Nodes interpret such messages and thus conduct the execution of services without consulting a centralised engine. However, nodes need to know explicitly which nodes to interact with and when, in a synchronous manner. A distributed workflow system based on mobile libraries playing the role of engines was presented in [41]. However, the details of the coordination are not described.

To increase loose coupling between services, a series of works relying on a shared space to exchange information between nodes have been proposed [94, 25, 71]. Following the Linda programming model [50], a tuplespace acts as a piece of memory shared by all interacting parties. Thus, using tuplespace for coordination, the execution of a part of a workflow within each node is triggered when tuples, matching the templates registered by the respective nodes, are present in the tuplespace. In the same vein, works such as KLAIM [76], propose a distributed architecture based on Linda where distributed tuplespaces store data and programs as tuples, allowing mobile computations by transferring programs from one tuple to another. However, in these works, the tuplespace is only used to store data information. The chemical programming model enhances the tuplespace model by providing a support for dynamics, and allows us to store both control and data information in the multiset.

The idea of using chemical programming to enact workflows autonomously was already explored in a few works [75, 38]. Thes works, however, remain very abstract, and give only very few clues on how to implement such approaches.

5.7 Conclusion

Through decentralisation, GinFlow allows to scale workflow coordination horizontally, and to get rid of unreliable orchestrators. GinFlow provides resilience mechanisms on top of its decentralised engine. GinFlow

is to my knowledge the only decentralised workflow engine to be available, with, at the time of writing, an active team, and a decently user-friendly usage..

Beyond the services it provides, GinFlow shows evidences that the chemical model offers nice abstractions, as a representative of distributed rule-based programming models, can help specifying such decentralised systems. Actually, this is even more striking when it comes to workflow adaptation, as we detail in the next chapter.

Chapter 6

Injecting Adaptiveness in (Decentralised) Workflow Execution

This chapter closes the technical part of this dissertation. It focus on supporting adaptiveness in the decentralised life-cycle of the workflow. More precisely, it describes how we extended GinFlow with the ability to adapt the shape of the workflow, i.e.,*replace a sub-part of the workflow by an alternate sub-workflow dynamically, transparently for the user and without having to replay the whole workflow. We explain how we again relied on the chemical model to develop this feature. Again, the design and implementation of these mechanisms was validated through a set of experiments run over the Grid'5000 platform.*

Joint work with:

- ⋆ Javier Rojas Balderrama (Postdoctoral researcher, Inria)
- ⋆ Matthieu Simonin (Research engineer, Inria)

6.1 The need for dynamically adapting workflows

Workflows are now one of the most common tools in a variety of science domains where computation-assisted experiments is the norm. In these developments, scientists are likely to explore different scenarios, design and test different workflows towards the final workflow structure that will actually produce helpful results. So the user is going back-and-forth between design and execution of the workflow, testing different workflow variants [52]. In many cases, the adequacy of one particular workflow shapes with regard to the scientist's goal can be established only after some intermediate output data are produced by some service of the workflow. Detecting this adequacy (or non-adequacy), adapting the workflow accordingly, and restarting the workflow can be time-consuming.

To help scientists with such a process, GinFlow is able to, given some user-defined function testing the intermediate output data, to stop the workflow as it was defined initially. Also, if provided with a workflow variant (a similar workflow typically differing from the initial one on a small fraction of the tasks), GinFlow will automatically *rewire* the workflow so as to switch from the initial workflow to the alternative one. GinFlow does this in decentralised settings and without restarting agents or replaying tasks that do not need to be replayed according to the rewiring. We distinguish two adaptation scenarios GinFlow supports: *cold* and *hot* adaptation.

6.1.1 *Cold* adaptation

The so-called *cold* adaptation corresponds to the scenario depicted in Figure 6.1 where the user (i) predefines statically what services in the workflow needs to be supervised / watched at run time (in beige in the picture). A simple condition to raise an exception can for instance be that the return code is not 0. More complex

conditions on the output, depending on the user's goal, can also be specified. Still at design time, the alternative workflow that will replace the supervised subworkflow is also specified by the user (in blue in the picture). ii) The workflow starts — running services are coloured in green, successfully completed ones are filled with black, and failed ones are coloured in red. iii) In the scenario depicted, the service in the middle of the supervised sequence triggers an exception. iv) GinFlow automatically rewire the workflow so as to remove watched services and inject the alternate sub-workflow into the workflow. v) The execution is resumed, but not from the beginning: already completed non-watched services are simply requested to send their results again to their new neighbours. vi) The execution completes according to the alternative plan.

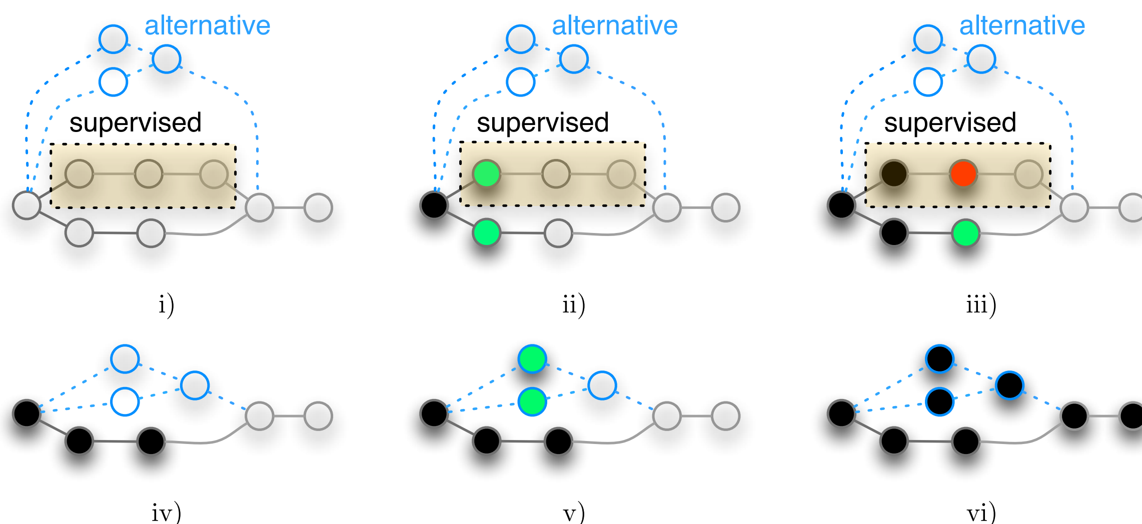


Figure 6.1: An example of cold adaptation.

Cold adaptation is useful if the user has already a good notion of what kind of problems may arise from its initial workflow and what alternate sub-workflow can replace the unsatisfying parts. It is the case for instance when the initial workflow is meant to obtain quickly results that night not have the required precision for the problem at stake, in which case the computation will have to go for a second, more insightful round.

6.1.2 *Hot* adaptation

Hot adaptation refers to a scenario where some services are supervised but the workflow developer has no precise idea on how to rewire the workflow prior to studying their output, and *a fortiori*, prior to starting the workflow. This scenario is illustrated in Figure 6.2. In this example, the initial workflow is defined with one particular service supervised (i). ii) The execution starts and iii) the supervised service fails. At this point, the user digs into the output of this service and decices what to do. Once the user is done defining an alternative workflow, he or she uses the GinFlow's API to program the alternate sub-workflow and how it should be integrated into the initial workflow. iv) The execution resumes, v) moves forward through the new sub-workflow and vi) completes.

This second scenario is very useful for a computational scientist who wants to explores the possible workflows to solve a problem without wasting time in useless re-executions of already completed tasks that do not change from one version of the workflow to the next one.

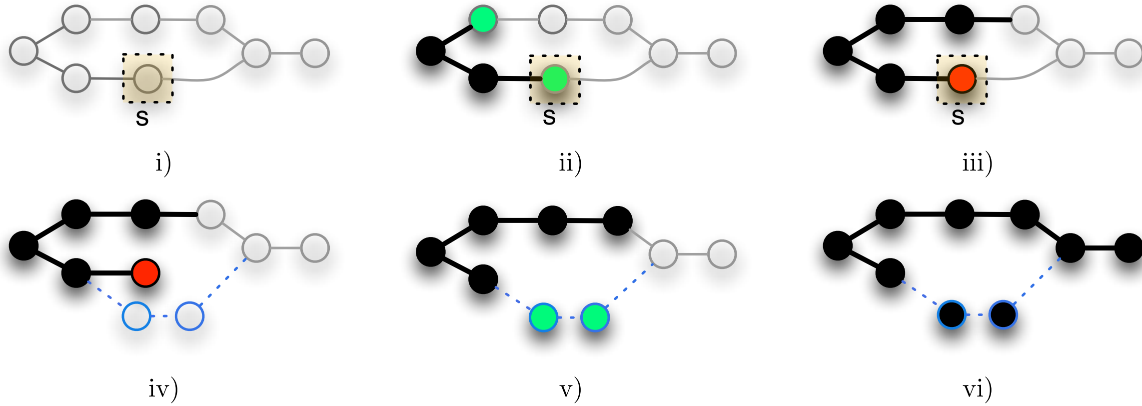


Figure 6.2: An example of hot adaptation.

6.2 HOCL-based adaptive workflows

Pursuing the idea of expressing the workflow execution coordination through HOCL rules as already used in Chapter 5, we now present how to express an HOCL abstract workflow that may be adapted at run time. We devise the rules needed to change the shape of the workflow on-the-fly (to switch from one workflow to another one). These rules are to be included in the HOCL program at starting time and will enact the reconfiguration needed, still in decentralised settings, and without the need for human intervention during enactment.

6.2.1 HOCL abstract adaptive workflow

Let us consider the workflow depicted in Figure 6.3. This is a simple sequence workflow where Task T_2 is considered as potentially producing unsatisfying results at run time. If T_2 actually *fails* producing adequate results, it is to be replaced by T_4. The corresponding HOCL$_{flow}$ program is given in Figure 6.3. The workflow provided is now organised into two parts. The first part (the first 3 lines) is the original workflow (without adaptiveness). The second part (Task T_{2b}), specifies a possible alternate behaviour.

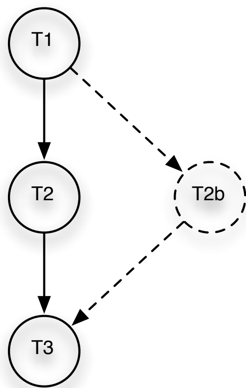


Figure 6.3: A simple adaptive workflow.

12.01 $\langle$
12.02 $\quad T_1 : \langle \text{SRC} : \langle\rangle, \text{DST} : \langle T_2 \rangle, \text{SRV} : \texttt{s1}, \text{IN} : \langle \texttt{input} \rangle \rangle,$
12.03 $\quad T_2 : \langle \text{SRC} : \langle T_1 \rangle, \text{DST} : \langle T_3 \rangle, \text{SRV} : \texttt{s2}, \text{IN} : \langle\rangle \rangle,$
12.04 $\quad T_3 : \langle \text{SRC} : \langle T_2 \rangle, \text{DST} : \langle\rangle, \text{SRV} : \texttt{s3}, \text{IN} : \langle\rangle \rangle,$

12.05 $\quad T_{2b} : \langle \text{SRC} : \langle T_1 \rangle, \text{DST} : \langle T_3 \rangle, \text{SRV} : \texttt{s2b}, \text{IN} : \langle\rangle \rangle$
12.06 $\rangle$

Figure 6.4: HOCL definition of an adaptive workflow.

Line 12.05 defines an alternate task named T_{2b}, which is triggered in case T_2 fails. This information is enough to generate all what is needed to inject T_{2b} if needed. The last remaining unknown parameter with

this description is what services are actually supervised. This is an information which is needed from the user. We will see in Section 6.3 how the user can actually provide this information.

6.2.2 Adaptation rules

The previous HOCL code example only expresses the abstract initial workflow and its alternate path. Note that this workflow can easily be inferred from the description of the workflow in another language (such as JSON). Still, this description needs to get extended to run over HOCL co-engines. The set of rules needed is illustrated in Figure 6.5.

```
13.01 trigger_adapt =
13.02     replace-one   T2 : ⟨RES : ⟨ERROR⟩, ω2⟩, T1 : ⟨ω1⟩, T3 : ⟨ω3⟩
13.03     by            T2 : ⟨ω2⟩, T1 : ⟨ADAPT, ω1⟩, T3 : ⟨ADAPT, ω3⟩

13.04 update_src =
13.05     replace-one   DST : ⟨⟩, ADAPT
13.06     by            DST : ⟨T2b⟩

13.07 update_dst =
13.08     replace-one   SRC : ⟨ωSRC⟩, IN : ⟨ωIN⟩, ADAPT
13.09     by            SRC : ⟨T2b⟩, IN : ⟨⟩
```

Figure 6.5: Adaptation rules.

Notice first that, in contrast with rules for traditional execution, these rules are specific to a particular pattern. It works only for the particular adaptation specified by the user and were generated out of it. The first rule initiates the adaptation phase: once a non-adequacy has been detected, the ERROR atom is injected into the T_2 solution, which triggers the trigger_adapt rule. This rule basically makes the ADAPT atom appear in the subsolutions pertained by the rewiring, in this specific case T_1 and T_3. The two subsequent rules are the specific rules that will actually rewire the workflow: The first one (update_src), upon the apparition of the ADAPT atom adds a new destination to the service which is the source of the adapted workflow. In our case, this rule will be placed at starting time in T_1's subsolution. The same goes with the other rule (update_dst), which replaces sources of the destination of the adapted workflow with T_{2b}. This last rule will be placed into T_3's subsolution at the beginning.

6.2.3 Execution

Now we have all the elements to build a concrete adaptive workflow. The full definition for this adaptive workflow is provided in Figure 6.6.

Let us review the execution of the workflow specified in Figure 6.6, which illustrates the initial state of the multiset, when the program starts. In this state, the only enabled rule is gw_setup in T_1. After it is applied, the gw_call rule can be applied as gw_setup initialised the PAR atom, required to apply gw_call. A first result is thus created, making gw_pass enabled, matching T_1 as the source, and T_2 as destination. Note that gw_pass is applied twice, filling the IN atom in T_2. The same process can start for T_2. For the sake of illustration, imagine T_2's invocation of `s2` does not produce an acceptable result. An ERROR atom appears in T_2. This triggers trigger_adapt, making the ADAPT atom appear in T_1 and T_3. The presence of this particular atom makes update_src and update_dst enabled. Their application adds a new destination in T_1 and updates the sources in T_3 so as to switch to the alternative scenario including T_{2b}. Given this new configuration, gw_pass is re-enabled on T_1 which sends T_1's result to T_{2b} which can invoke `s2b`. When the result of T_{2b} enters T_3's subsolution (using gw_pass), the workflow completes by triggering gw_callon `s3`.

```
14.01 ⟨
14.02   gw_pass,
14.03   T1 : ⟨SRC : ⟨⟩, DST : ⟨T2⟩, SRV : s1, IN : ⟨input⟩, gw_setup, gw_call, update_src⟩,
14.04   T2 : ⟨SRC : ⟨T1⟩, DST : ⟨T3⟩, SRV : s2, IN : ⟨⟩, gw_setup, gw_call, trigger_adapt⟩,
14.05   T3 : ⟨SRC : ⟨T2⟩, DST : ⟨⟩, SRV : s3, IN : ⟨⟩, gw_setup, gw_call, update_dst⟩,
14.06   T2b : ⟨SRC : ⟨T1⟩, DST : ⟨T3⟩, SRV : s2b, IN : ⟨⟩⟩
14.07 ⟩
```

Figure 6.6: The concrete adaptive workflow.

6.2.4 Generalising

The previous example dealt only with one task being replaced by another one. GinFlow supports more complex adaptations. Let us have a look on what is actually possible to do online. Would it be possible to replace any part of the workflow by any other graph? Let us denote $G = (V, E)$ the DAG of the initial workflow, where V represents the tasks, and E the (oriented) dependencies. Let us denote $V_S \subseteq V$ the set of supervised tasks. We assume that the graph $G_S = (V_S, E_S)$ which includes all vertices connecting two nodes in V_S is connected. A first constraint is that there must be a single external node to which all dependencies going out from V_S point to, *external* meaning not in G_S.

More formally, let us define the set $V_{S'}$ of nodes, *at the edge of* G_S:

$$V_{S'} = \{u \in V_S \mid \exists v \in V : \vec{uv} \in E \wedge v \notin V_S\}$$

Let us define Ext_u the set of external destinations of nodes in $V_{S'}$:

$$Ext_u = \{v \mid \vec{uv} \in E \wedge u \in V_S \wedge v \notin V_S\}$$

We need to satisfy the two following constraints:

$$\begin{cases} \forall u, v \in V_{S'}, Ext_u = Ext_v \\ \forall u \in V_{S'}, |Ext_u| = 1 \end{cases} \tag{6.1}$$

These constraints are illustrated in Figure 6.7. The set of services supervised in a) and b) satisfies the constraint, but not the one in c): in this last case, two services with diverging external outgoing dependencies are supervised. Allowing such an adaptation could lead to replace a subworkflow whose partial output has been sent forward in the workflow, bringing about potential inconsistencies. The alternate subworkflow can also be seen as a connected DAG, say $G_R = (V_R, E_R)$. As it replaces the set of supervised services, it needs to get wired so as its outgoing edges are connected to the service which lost its incoming dependencies with the removal of the services in V_S.

Consequently, V_R is similar to V_S in definition. The two constraints expressed by Formula 6.1 G_S are also required for G_R. Moreover, the constraint over both V_R and V_S is that they are wired to the same external outgoing task. Formally, we add Constraint 6.2 to Constraints 6.1 on both G_S and G_R:

$$\forall u \in V_{S'}, v \in V_{R'}, Ext_u = Ext_v \tag{6.2}$$

This last constraint is illustrated in Figure 6.7 (d): the alternate workflow has an outgoing link to a node which is not the same as the external outgoing link of the supervised service to be replaced. This adaptation is thus not *valid*. Enabling such an adaptation could lead to potential problems: If the faulty workflow already produced the output for one of its outgoing links at failure time, this prior-to-failure result will propagate through the rest of the workflow in spite of the computation being replayed through the alternative workflows, possibly leading to conflicting set of data later in the workflow. While this problem could be solved by stopping the workflow, cleaning the execution environment, redesigning the workflow, and restarting it, this falls out of the scope of the paper, which focus on what can be done *online*.

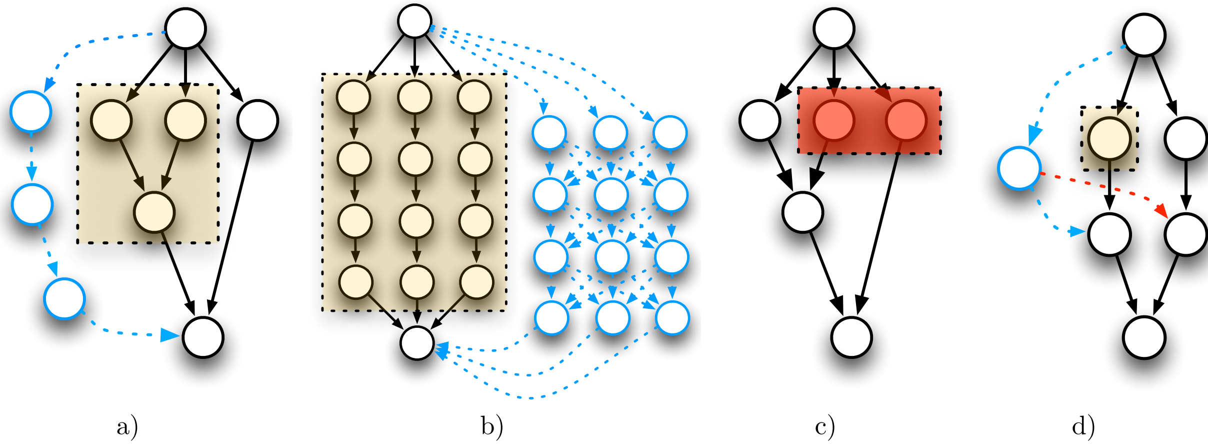


Figure 6.7: Correct and incorrect adaptation scenarios.

6.3 Adaptiveness in GinFlow

Chapter 5 presented the architecture, features and client interfaces of GinFlow. Let us have a quick view on how the adaptiveness feature has been implemented into GinFlow and how a workflow designer can use it.

6.3.1 Implementation

As explained in Chapter 5 (Section 5.4.1), the workflow management rules are injected into GinFlow prior to deploying the agents. The same goes for the adaptation rules in case of a *cold* adaptation. Still, as mentioned earlier, the rules to inject are specific to the adaptation pattern: they are generated from the specification of the supervised subworkflow and the alternative subworkflow specified by the user. The generation is done beforehand, prior to the injection.

Recall the gw_pass rule, mentioned in Chapter 5, which spans several subsolutions, and thus requires communication to push data from one source SA to one destination SA. The same goes with the trigger_adapt rule. The agent raising the exception will make the ADAPT atom appear inside the subsolutions of the agents responsible for the services to be adapted (according to the adaptation plan provided by the user) by sending messages to them.

The implementation allows both *hot* and *cold* adaptation presented in Section 6.1. In case of hot adaptation, GinFlow bascially suspends the execution of the workflow when the exception is raised, reflecting that the output of some supervised service was not adequate. In this situation, one may choose to update the workflow by submitting an adaptation plan through the API after the execution was suspended. GinFlow then deploys alternative SAs, send them their initial descriptions and inject adaptation rules to the SAs managing services taking part in the adaptation. The following section shows how the user can specify the adaptation pattern he or she needs.

6.3.2 Client interfaces

The key aspects of GinFlow's client interface of GinFlow was presented in Section 5.4.4. We now focus on how the workflow designer will use GinFlow's interfaces in case of an adaptation.

Adaptiveness may again be specified using either a JSON file or through the Java API. (The GinFlow Java API provides constructs similar to he ones presented in Section 5.4.4, but for adaptiveness). Let us have a look at the example written in JSON that specifies an adaptive workflow is shown in Listing 2. The listing shows two parts. The first one is the JSON definition of the workflow depicted in Figure 6.3. The

second part, on the right, expresses the alternate subworkflow (2*b* replacing 2) along with the rewiring of the graph: firstly, we need to update the source of the new workflow (Task 1) with its new destination (Task 2b). Secondly, we need to update the destination of the new workflow (Task 3) with its new source (Task 2b).

If the user does not know *a priori* what alternate sub-workflow is relevant, following a *hot* adaptation scenario, the initial JSON file sent to GinFlow needs to contain only the information about tasks 1, 2 and 3. When GinFlow suspends the execution, the user will have to write and send the difference between this initial description and the JSON file given in Listing 2: the right (*rebranching*) part.

```
{
  "name"    : "wf-adapt",
  "services": [
  {
    "name"        : ["1"],
    "srv"         : ["echo"],
    "in"          : ["1"],
    "src_control" : [],
    "dst_control" : ["2"]
  },
  {
    "name"       : ["2"],
    "srv"        : ["notfound"],
    "in"         : ["2"],
    "dst_control": ["3"],
    "src_control": ["1"]
  },
  {
    "name"        : ["3"],
    "srv"         : ["echo"],
    "in"          : ["3"],
    "src_control" : ["2"],
    "dst_control" : []
  },
```

```
  {
    "name": ["2b"],
    "srv" : ["echo"],
    "in"  : ["alt!"],
    "src" : ["1"],
    "dst" : ["3"]
  }],
  "rebranchings": [{
    "supervised": ["2"],
    "updateSrc" : {"1": ["2b"]},
    "updateDst" : {"3": ["2b"]}
  }]
}
```

Listing 2: A JSON file defining an adaptive workflow.

6.4 Experimental evaluation

With our adaptiveness mechanisms implemented into GinFlow, we wanted to assess its performance. Performance in this context may not be the primary concern. For instance, in the context of *hot* adaptiveness, the time it takes to the programmer to redesign its workflow makes it likely superfluous to over-optimize the restarting phase. Still, this restarting duration matters. In order to evaluate the speed of the restarting phase, we run experiments with the same experimental set-up as in Section 5.5.

We executed both diamond-shaped workflow flavours in adaptive scenarios. Initially, we performed a regular execution (without adaptiveness) as a reference using square configurations (i.e., with $h = v$). Then, we executed the same workflows, but raising an execution exception on the last service of the mesh, and replacing the whole body of the diamond on-the-fly. The goal is to observe the effect of adaptiveness after the execution of the most part of the workflow, and compare it to a complete re-execution of the workflow. These executions are performed in three different scenarios: (1) replacing a simply-connected diamond body by another diamond with the same shape, (2) replacing a simple-connected body by another which is fully-connected, in a similar way to the schema shown in Figure 6.7(b), and (3) replacing a fully-connected workflow by another one which is simply-connected. These scenarios were also designed to evaluate the cost of adaptiveness when multiple links need to be reconfigured in the outermost (first and last) services of the diamond workflow and that almost the whole workflow is to be replaced.

Figure 6.8 exhibits, for each scenario, the ratio between the execution time with adaptiveness and the total time it would have take to restart the workflow after the error. A ratio below 1 shows it is faster to not restart a workflow after an error but let the adaptiveness occurs. In the first scenario, the ratio never exceeds

1. In other words, adaptiveness, in the worst case where every service (except the final one) needs to be replayed, is quicker than a complete re-execution whose time is at least twice the execution of the workflow without adaptiveness. The second scenario shows that for configurations bigger than 1×1 the adaptiveness ratio remains in the same order, below 1, in spite of the significant number of links to reconfigure and the involved services (up to 21×21). Finally, the third scenario shows that the ratio remains also constant and below 1.

6.5 Adapting the workflow shape in literature

While many workflow manager systems initiatives achieve some enactment flexibility at the infrastructure level, through for instance re-submissions strategies [67], pilot jobs [28] and task replications [48]), other works [99, 58] explicitly deal with adaptiveness for scientific workflows at the application level. In [99], authors propose two patterns to manage the exception handling based on the Reference Nets-within-Nets formalism: propagation and replacement. These mechanisms for dynamically adapting the workflow structure at run time rely on a workflow representation model where the original workflow scenario is mixed with the alternative path. FireWorks [58] also provides a dynamic feature to add new components to the workflow while it is running. This workflow manager targets the same use-cases failures as GinFlow but in a less-general way in the sense that they support only what we refer to as *cold adaptation*.

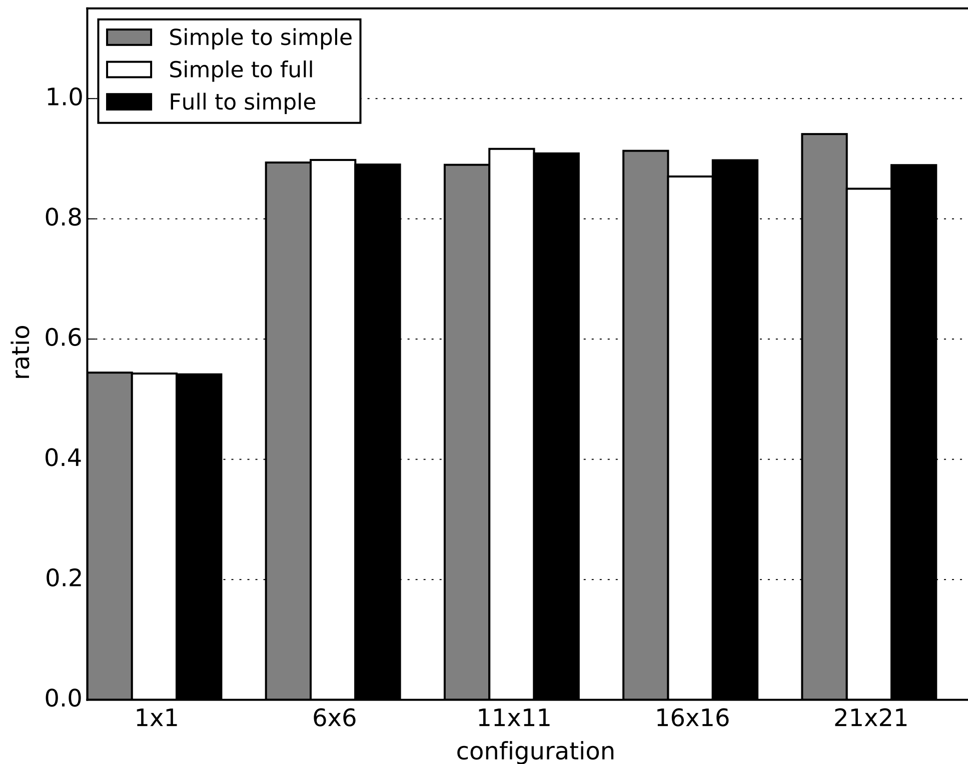


Figure 6.8: With-adaptiveness-over-without-adaptiveness ratio.

6.6 Conclusion

We designed an adaptiveness mechanism aiming at helping the workflow designer dealing with the uncertain nature of any computational science venture. The feature designed, developed and experimented allows the workflow designer to specify alternate subworkflows either prior to running the workflow or during the workflow, when the execution manager raises an exception. The adaptation from the initial workflow to the alternative one will get enacted transparently for the user.

Specified using HOCL and intended to run in decentralised settings, this mechanism was implemented in GinFlow and its extra cost at run time was shown to be zero most of the time for diamond-shaped workflows made of up to 443 services out of which 441 are adapted and replaced by another set of 441 services.

Conclusion

An exploration ...

The initial motivation behind this series of works as well as our early contributions have been presented only in Chapter 5. They were directed towards the proposal of an architecture to decentralise the coordination of composite services following the observation that, as stated in Chapter , component models, which have been recently promising models to compose services and execute them in decentralised settings, are not built upon abstractions that easily support loose coupling and adaptation.

In our quest for alternative models to enact and adapt composite services in decentralised settings, as it has been detailed throughout this dissertation, HOCL acted as both a source of inspiration and an implementation tool. I do not argue that HOCL, and more generally, chemically-inspired programming styles, and even more generally, higher-order rule-based models, are the definite way to program service compositions. This series of work appears to me as being part of an exploration that could be summarized as follows: *It appears that the abstractions conveyed by HOCL could help in our quest for programming styles for large scale adaptive platforms, let us see where it leads if we assume it is true.* The question whether it is actually the *right* way to go is beyond the scope of this work, and anyway, I believe, not a particularly relevant question.

The fact that this work is more exploratory than a large proportion of research work carried out in workflow computing constitutes both its weakness and its strength. This is a weakness mostly because there is no precise metric that could establish whether our approach outperforms others. Still, I am not sure whether it would actually makes sense to compare our approach with others performance-wise. Also, our contributions are a bit more difficult to position in the state of the art, in the sense that they are not the result of incremental steps only, but the outcome of more alternative research. While alternative *ventures* in research might carry higher risks of failure, in particular due to their difficulty of positioning, they are more likely to bring a fresh eye on to some problem at stake.

I believe the out-of-the-mainstream aspect of this work constitutes a strength, because HOCL *actually* makes the workflow shape adaptation possible. I am not saying that there is no other way to achieve it. I just think HOCL inspired us on *how* to do it. More generally, this work conveys the idea that these models can be of great help when there is a need for specifying adaptive systems, and we wanted to show it through a concrete example, from the design to some experimental evaluation. May it be inspirational to other researchers.

Exploring alternative approaches come with nice opportunities to revisit classical problems. This is what happened to us when trying to implement a distributed HOCL runtime. We had to revisit the classical resource allocation problem. So, it was a double inspiration. Firstly, HOCL helped us tackling the decentralised adaptive workflow execution problem. Secondly, it allowed us to revisit some classical distributed systems problems, adding (or relaxing) constraints brought about by its implementation in large distributed settings. Note that this idea of having protocols that dynamically and locally adapt back-and-forth between an optimistic approach and a pessimistic approach can be seen as an application of the vision conveyed by autonomic computing since a few years.

... with concrete outcomes ...

Relying over out-of-the-mainstream concepts did not prevent us to make a few concrete contributions. Even if they were extensively discussed in the *technical* chapters of this dissertation, let us summarize them and refer the reader to their related publications. These contributions can be classified into two categories.

The first category of contributions gathers results around the ability to execute workflows in decentralised settings in an adaptive manner, subjects of Chapters 5 and 6. They are contributions to the field of workflow management, with ideas echoing the autonomic computing paradigm. This is where HOCL was used as an inspiration, to propose an architecture for a distributed workflow engine, to specify it, and to build a software prototype of it, initially called *HOCL-TS*. The initial proposal for an architecture and the specification of an HOCL-based execution engine was made in [45]. Some early experiments testing the viability of the approach conducted through the deployment of HOCL-TS are reported in [46] and [47]. The extension of these works with adaptiveness is presented in [87], as well as the refactoring, cleaning and extension of HOCL-TS into GinFlow, a now ready-to-be-used workflow execution manager whose code has been released in 2015.

The second category of contributions deals with the study of the model itself. On one hand, its complexity has been more precisely captured — the subject of Chapter 2 and paper [19] —, and on the other hand, its implementation in distributed settings has been studied, — matter of Chapters 3 and 4. In this last stream of research, the contributions are i) the proposal of an adaptive resource allocation algorithm — presented in detail in [17] —, ii) the design of two distributed environments supporting the execution of programs written in the *chemical* style, lifting a barrier towards the use of such models at large scale. The design, implementation and experimental validation of the hierarchical and flat reactors are presented in [79] and [78], respectively.

... to be continued ...

GinFlow has recently been equipped with a graphical interface to specify workflows to be submitted to the workflow engine. It is particularly useful to deal with *hot* adaptation.

A planned action for GinFlow is to find more use cases for the adaptiveness feature. In particular, several cosmology codes and their particular usage have been identified as potential benefiters of GinFlow. The first one is Gadget [95], a code for cosmological simulations of structure formation which requires the cosmologist to run an entire simulation and analyse the results to start a second one with the parameters updated according to the results of the first one. This kind of refinement process could benefit from the GinFlow adaptation mechanisms. Another cosmology project which could leverage GinFlow's adaptiveness is the Supernova Factory code, aiming at measuring the expansion history of the Universe and explore the nature of Dark Energy [5].

Note that GinFlow has been interfaced with the Tigres workflow runtime system [84], developed at Lawrence Berkeley National Lab (LBL). Tigres aims first at providing a set of programming abstractions to ease the integration of workflow patterns into scientific codes. The supported templates are sequences, parallel splits, forks and joins. The goal of the binding between Tigres and GinFlow was to extend Tigres' potential execution platforms through decentralisation. In this binding, each template is sent to GinFlow, which executes it and sends the output of the template back to the Tigres runtime, which can then move forward in the execution. Details of the design of this bridge can be found in [86].

... for the people

The application space where the programming model conveyed by workflows is useful is much broader than science. As we mentioned at the beginning of this document, the early efforts towards programming service compositions came from industry. The workflow paradigm is now moving to the every day life, with the

apparition of tools helping people to build their own compositions (or *recipes*) involving services such as emailing, micro-blogging, social networking, and storage management.

IFTTT (*If that, then this*) is a tool [3] that provides a nice graphical interface to create and use these *recipes* such as *check the weather, and if it is fine, notify me and find me the closest ice-cream parlor.* In April 2016, Microsoft Flow [4] which provides similar features, went live. Workflows that can be defined with Microsoft Flow includes many (cloud-based) services such as Slack, Github and Dropbox. While such a tool is very appealing, it comes with the necessity to trust the coordinating third-party: the Microsoft flow server, on which our workflows are stored and managed. (The first mandatory thing to do when trying Microsoft Flow is to open an account on the Microsoft Flow server). This leads us back to our initial concerns with centralisation. What these trusted third-parties do with our data is never mentioned when one opens an account.

Until recently, many services used in every day life worked due to the presence of a trusted third party. Two common examples are banking and notary certification. In the case of banking, the need is to avoid double spending of the same money and thus certify the transaction. Notary certification is useful to ensure that a document is *real* in the sense that it was the same when some contract it represents was signed. These two examples actually rely on one need: time-stamping objects (so they can be ordered in time).

Recently, the blockchain changed this apparently frozen situation by proposing a way to certify transactions (and more generally *objects*) in a decentralised way removing the need for a trusted third-party authority. The blockchain was initially proposed in the context of banking, as a way to replace the central authority (the banker), by a decentralised cryptographic proof. This allowed the rise of cryptocurrencies, the most known among them being the bitcoin [74], for which these concepts were initially proposed.

The validity of a transaction relies on a proof-of-work. Before it is included into the blockchain (*i.e.,* the official ledger), a transaction is first inserted into a block of transactions along with other new transactions. Then, a *nonce* needs to be found such that it satisfies a hard-to-obtain property: a cryptographic function applied on the block and this nonce must return a number which is lower than a given threshold. Finding such a nonce requires an amount of computational work which grows inversely to the value threshold. Once it is found by some node, the block is inserted into the blockchain locally on this node and then sent over the network as the new, up-to-date *ledger*. In case of several ledgers coexisting in the network, the longest one is systematically favoured over the smaller ones. This means that an attacker who wants to disseminate a modified ledger needs to build proof-of-works quicker than the *honest* cooperating nodes in the network. Consequently, as long as the combined computational power of the honest nodes predominates, the probability of a successful attack is very low.

The threats on privacy brought about by all-powerful centralised services offering social networking functionalities are multiple. Note that, in this context, centralised does not so much refer to the centralisation of the infrastructure than to the fact that a single entity controls it. (Facebook's few data centers are actually distributed across the world.) Trusting such entities to store and expose our data to our *friends* without using them for their own sake, while this the service is free, appears to be quite irrational, or at least naive. The numerous breaches on personal data recently experienced[1] highlight the need for alternate architectures providing the same kind of functionalities. This is where decentralised approaches can help. Recently, in the context of social networking, several new platforms and approaches have been proposed to try to fill this gap. A first step towards decentralisation has been achieved by platforms such as Diaspora [1], through a federation of servers, and SuperNova [91] which relies on super-peers to store users' data. Fully decentralised approaches were also proposed in the literature, such as Safebook [36], ProofBook [20] and SOUP [62]. These approaches ensures different levels of privacy, anonymity and security in the network, based on decentralisation, cryptographic hashing and proof-of-works. In a recent joint work with Arnaud

[1] Go to `https://en.wikipedia.org/wiki/Data_breach#Major_incidents` for a list.

Fabre and Marin Bertier, we explored these techniques to propose a decentralised online social network with an emphasis on ensuring anonymity and confidentiality while providing a high level of availability on the data. Our work also allows users to belong to several groups and share some content with some group while completely hiding it to the others [43].

As can be said about the blockchain, our approach does not need *trust* to work. More generally, decentralisation combined with cryptography in a broad sense seems to pave the way for privacy-preserving service orchestration at large, for the people.

Bibliography

[1] Diaspora*. https://joindiaspora.com/. Retrieved December 2016.

[2] Freepastry. http://www.freepastry.org. Retrieved December 2016.

[3] IFTTT. https://ifttt.com. Retrieved December 2016.

[4] Microsoft flow. https://flow.microsoft.com. Retrieved December 2016.

[5] G. Aldering, G. Adam, P. Antilogus, P. Astier, R. Bacon, S. Bongard, C. Bonnaud, Y. Copin, D. Hardin, F. Henault, D. Howell, J.-P. Lemonnier, J.-M. Levy, S. Loken, P. Nugent, R. Pain, A. Pecontal, E. Pecontal, S. Perlmutter, R. Quimby, K. Schahmaneche, G. Smadja, and W. Wood-Vasey. Overview of the Nearby Supernova Factory. *SPIE Proceedings Series*, 4836:61–72, 2002.

[6] W. R. Ashby. *Design for a Brain: The Origin of Adaptive Behavior*. Chapman and Hall, 2nd edition, 1960.

[7] R. J. Bagley and D. J. Farmer. Spontaneous Emergence of a Metabolism. In C. G. Langton, C. Taylor, D. J. Farmer, and S. Rasmussen, editors, *Artificial Life II*, pages 93–140, Redwood City, CA, 1992. Addison-Wesley.

[8] J. Banâtre and D. Le Métayer. The GAMMA Model and its Discipline of Programming. *Sci. Comput. Program.*, 15(1):55–77, 1990.

[9] J.-P. Banâtre, A. Coutant, and D. Le Métayer. A Parallel Machine for Multiset Transformation and its Programming Style. *Future Gener. Comput. Syst.*, 4(2):133–144, Sept. 1988.

[10] J.-P. Banâtre, P. Fradet, and D. Le Métayer. *Multiset Processing: Mathematical,Computer Science, and Molecular Computing Points of View*, chapter Gamma and the Chemical Reaction Model: Fifteen Years After, pages 17–44. Springer Berlin Heidelberg, 2001.

[11] J.-P. Banâtre, P. Fradet, and Y. Radenac. Generalised Multisets for Chemical Programming. *Mathematical Structures in Computer Science*, 16, 2006.

[12] J.-P. Banâtre and D. Le Métayer. Programming by Multiset Transformation. *Commun. ACM*, 36(1):98–111, 1993.

[13] A. Barros, M. Dumas, and P. Oaks. A Critical Overview of the Web Services Choreography Description Language. *BPTrends*, Mar. 2005.

[14] F. Baude, D. Caromel, C. Dalmasso, M. Danelutto, V. Getov, L. Henrio, and C. Pérez. GCM: a Grid Extension to Fractal for Autonomous Distributed Components. *Annales des Télécommunications*, 64(1-2):5–24, 2009.

[15] BEA, IBM, Interface21, IONA, Oracle, SAP, Siebel, and Sybase. Service Component Architecture - Building Systems using a Service Oriented Architecture - A joint whitepaper. White paper, Nov. 2005.

[16] G. Berry and G. Boudol. The Chemical Abstract Machine. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'90, pages 81–94, New York, USA, 1990. ACM.

[17] M. Bertier, M. Obrovac, and C. Tedeschi. Adaptive Atomic Capture of Multiple Molecules. *J. Parallel Distrib. Comput.*, 73(9):1251–1266, Sept. 2013.

[18] M. Bertier, M. Perrin, and C. Tedeschi. On the Complexity of Concurrent Multiset Rewriting. Research Report RR-8408, Inria, Dec. 2013. Available at: `https://hal.inria.fr/hal-00912554`.

[19] M. Bertier, M. Perrin, and C. Tedeschi. On the Complexity of Concurrent Multiset Rewriting. *International Journal of Foundations of Computer Science*, 27(1):67–83, 2016.

[20] S. Biedermann, N. P. Karvelas, S. Katzenbeisser, T. Strufe, and A. Peter. ProofBook: An Online Social Network Based on Proof-of-Work and Friend-Propagation. In *Proceedings of the 40th International Conference on Current Trends in Theory and Practice of Computer Science (SOSFEM 2014)*, pages 114–125. Springer, 2014.

[21] W. Binder, I. Constantinescu, and B. Faltings. Decentralized Orchestration of Composite Web Services. In *Proceedings of the 4th IEEE International Conference on Web Services*, Chicago, Sept. 2006.

[22] H. Bouziane, C. Pérez, and T. Priol. A Software Component Model with Spatial and Temporal Compositions for Grid Infrastructures. In *14th International Euro-Par Conference on Parallel Processing (Euro-Par 2008)*, pages 698–708, Las Palmas de Gran Canaria, Spain, Aug. 2008.

[23] I. Brito. Synchronous, Asynchronous and Hybrid Algorithms for DisCSP. In M. Wallace, editor, *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming (CP 2004)*, volume 3258 of *Lecture Notes in Computer Science*, pages 791–791. Springer, 2004.

[24] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J. Stefani. The FRACTAL Component Model and its Support in Java. *Softw., Pract. Exper.*, 36(11-12):1257–1284, 2006.

[25] P. A. Buhler and J. M. Vidal. Enacting BPEL4WS Specified Workflows with Multiagent Systems. In *Proceedings of the 2nd Workshop on Web Services and Agent-Based Engineering*, July 2004.

[26] K. Candan, J. Tatemura, D. Agrawal, and D. Cavendish. 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.

[27] F. Cappello, E. Caron, M. Daydé, F. Desprez, Y. Jégou, P. Primet, E. Jeannot, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, B. Quetier, and O. Richard. Grid'5000: a Large Scale and Highly Reconfigurable Grid Experimental Testbed. *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, 2005.

[28] A. Casajus, R. Graciani, S. Paterson, and A. Tsaregorodtsev. DIRAC pilot framework and the DIRAC Workload Management System. *Journal of Physics: Conference Series*, 219(6), 2010.

[29] K. M. Chandy and J. Misra. The Drinking Philosophers Problem. *ACM Transactions on Programming Languages and Systems*, 6(4):632–646, 1984.

[30] A. Chatalic, M. Bertier, and C. Tedeschi. Efficient Execution of Chemically-Inspired Programs Using Gossip Protocols. internship report, Ecole normale supérieure de Rennes, 2013. Retrieved in December 2016 at `http://perso.eleves.ens-rennes.fr/~achatali/res/rapport_stage_mit1.pdf`. In French.

[31] D. Cohen and J. M. Filho. Introducing a Calculus for Higher-Order Multiset Programming. In *First International Conference on Coordination Languages and Models (COORDINATION'96)*, pages 124–141, Cesena, Italy, Apr. 1996.

[32] D. Conte, P. Foggia, C. Sansone, and M. Vento. Thirty Years of Graph Matching in Pattern Recognition. *International Journal of Pattern Recognition and Artificial Intelligence*, 18(3):265–298, 2004.

[33] S. A. Cook. The Complexity of Theorem-Proving Procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing (STOC)*, pages 151–158, Shaker Heights, USA, May 1971.

[34] S. B. Cooper. *Computability Theory.* Chapman and Hall, 2003.

[35] C. Creveuil. *Research Directions in High-Level Parallel Programming Languages*, chapter Implementation of Gamma on the connection machine, pages 219–230. Springer Berlin Heidelberg, 1992.

[36] L. A. Cutillo, R. Molva, and T. Strufe. Safebook: A Privacy-Preserving Online Social Network Leveraging on Real-Life Trust. *IEEE Communications Magazine*, 47(12):94–101, 2009.

[37] A. Denis, C. Pérez, T. Priol, and A. Ribes. Padico: A Component-Based Software Infrastructure for Grid Computing. In *17th International Parallel and Distributed Processing Symposium (IPDPS'03)*, Nice, France, Apr. 2003.

[38] C. Di Napoli, M. Giordano, Z. Németh, and N. Tonellotto. Adaptive Instantiation of Service Workflows Using a Chemical Approach. In *Proceedings of the 16th International Euro-Par Conference on Parallel Processing*, Ischia, Italy, Aug. 2010.

[39] P. Dittrich, J. Ziegler, and W. Banzhaf. Artificial chemistries – a Review. *Artificial Life*, 7:225–275, June 2001.

[40] A. Doniec, S. Piechowiak, and R. Mandiau. A DisCSP Solving Algorithm Based on Sessions. In I. Russell and Z. Markov, editors, *18th International Florida Artificial Intelligence Research Society Conference (FLAIRS)*, pages 666–670. AAAI Press, 2005.

[41] P. Downes, O. Curran, J. Cunniffe, and A. Shearer. Distributed Radiotherapy Simulation with the Webcom Workflow System. *Int. J. High Perform. Comput. Appl.*, 24:213–227, 2010.

[42] T. B. Downing. *Java RMI: Remote Method Invocation.* IDG Books Worldwide, 1st edition, 1998.

[43] A. Fabre, M. Bertier, and C. Tedeschi. Distributed Algorithms for Personal Data Management. Master thesis, Université de Rennes 1, 2016. In french.

[44] H. Fernández. *Flexible Coordination Based on the Chemical Metaphor for Service Infrastructures.* PhD thesis, Université de Rennes 1, May 2012.

[45] H. Fernández, T. Priol, and C. Tedeschi. Decentralized Approach for Execution of Composite Web Services Using the Chemical Paradigm. In *8th International Conference on Web Services (ICWS)*, pages 139–146, 2010.

[46] H. Fernández, C. Tedeschi, and T. Priol. Rule-driven Service Coordination Middleware for Scientific Applications. *Future Generation Computer Systems*, 35:1–13, 2014.

[47] H. Fernández, C. Tedeschi, and T. Priol. A Chemistry-Inspired Workflow Management System for a Decentralized Workflow Execution. *IEEE Transactions on Services Computing*, 9(2):213–226, 2016.

[48] R. Ferreira da Silva, T. Glatard, and F. Desprez. Self-Healing of Workflow Activity Incidents on Distributed Computing Infrastructures. *Future Generation Computer Systems*, 29(8):2284–2294, 2013.

[49] W. Fontana and L. W. Buss. The Arrival of the Fittest: Toward a Theory of Biological Organization. *Bulletin of Mathematical Biology*, 56(1):1–64, 1994.

[50] D. Gelernter. Generative Communication in Linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.

[51] J. Giavitto, O. Michel, and A. Spicher. Spatial Organization of the Chemical Paradigm and the Specification of Autonomic Systems. In *Software-Intensive Systems and New Computing Paradigms - Challenges and Visions*, volume 5380 of *Lecture Notes in Computer Science*, pages 235–254. Springer, 2008.

[52] Y. Gil, E. Deelman, M. H. Ellisman, T. Fahringer, G. Fox, D. Gannon, C. A. Goble, M. Livny, L. Moreau, and J. Myers. Examining the Challenges of Scientific Workflows. *IEEE Computer*, 40(12):24–32, 2007.

[53] T. Glatard, J. Montagnat, D. Lingrand, and X. Pennec. Flexible and Efficient Workflow Deployment of Data-Intensive Applications On Grids With MOTEUR. *Int. J. High Perform. Comput. Appl.*, 22(3):347–360, 2008.

[54] S. Grumbach and F. Wang. Netlog, a Rule-Based Language for Distributed Programming. In *12th International Symposium on Practical Aspects of Declarative Languages (PADL)*, volume 5937 of *Lecture Notes in Computer Science*, pages 88–103, Madrid, Spain, Jan. 2010. Springer.

[55] F. Hayes-Roth. Rule-Based Systems. *Communications of the ACM*, 28(9):921–932, Sept. 1985.

[56] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, Boston, USA, Apr. 2011.

[57] IBM. An Architectural Blueprint for Autonomic Computing. White paper, June 2005.

[58] A. Jain, S. P. Ong, W. Chen, B. Medasani, X. Qu, M. Kocher, M. Brafman, G. Petretto, G.-M. Rignanese, G. Hautier, D. Gunter, and K. A. Persson. FireWorks: a Dynamic Workflow System Designed for High-Throughput Applications. *Concurrency and Computation: Practice and Experience*, 27(17):5037–5059, 2015.

[59] M. Jelasity, A. Montresor, and O. Babaoglu. Gossip-based Aggregation in Large Dynamic Networks. *ACM Trans. Comput. Syst.*, 23(3):219–252, Aug. 2005.

[60] K. Jensen and L. M. Kristensen. *Coloured Petri Nets: Modelling and Validation of Concurrent Systems.* Springer, 1st edition, 2009.

[61] A.-M. Kermarrec and M. van Steen. Gossiping in Distributed Systems. *SIGOPS Oper. Syst. Rev.*, 41(5):2–7, Oct. 2007.

[62] D. Koll, J. Li, and X. Fu. Soup: an Online Social Network by the People, for the People. In *15th International ACM/IFIP/USENIX Middleware Conference*, pages 193–204. ACM, 2014.

[63] H. Kuchen and K. Gladitz. Parallel Implementation of Bags. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, FPCA'93, pages 299–307, New York, USA, 1993. ACM.

[64] J. Larrosa and G. Valiente. Constraint Satisfaction Algorithms for Graph Pattern Matching. *Mathematical. Structures in Comp. Sci.*, 12(4):403–422, Aug. 2002.

[65] D. Le Métayer. Higher-Order Multiset Programming. In *Proceedings of the DIMACS workshop on specifications of parallel algorithms*, DIMACS series in Discrete Mathematics. American Mathematical Society, 1994.

[66] H. Lin, J. Kemp, and P. Gilbert. Computing Gamma Calculus on Computer Cluster. *International Journal of Technology Diffusion*, 1(4):42–52, 2010.

[67] D. Lingrand, J. Montagnat, and T. Glatard. Modeling User Submission Strategies on Production Grids. In *Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing*, Munich, Germany, June 2009.

[68] J. W. Lloyd. Practical Advantages of Declarative Programming. In *Proceedings of the 1994 Joint Conference on Declarative Programming (GULP-PRODE'94)*, pages 18–30.

[69] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao. Scientific Workflow Management and the Kepler System. *Concurrency Comput.: Pract. Exper.*, 18(10):1039–1065, 2006.

[70] N. A. Lynch, D. Malkhi, and D. Ratajczak. Atomic Data Access in Distributed Hash Tables. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS'01, pages 295–305, London, UK, 2002. Springer-Verlag.

[71] D. Martin, D. Wutke, and F. Leymann. A Novel Approach to Decentralized Workflow Enactment. In *Proceedings of the 12th International IEEE Enterprise Distributed Object Computing Conference*, Munich, Germany, Sept. 2008.

[72] B. T. Messmer and H. Bunke. Subgraph Isomorphism in Polynomial Time. Research report, Institut fur Informatik und angewandte Mathematik, University of Bern, Switzerland, 1995.

[73] R. A. Micillo, S. Venticinque, N. Mazzocca, and R. Aversa. An Agent-Based Approach for Distributed Execution of Composite Web Services. In *Proceedings of the 17th IEEE International Workshops on Enabling Technologies*, Rome, Italy, June 2008.

[74] S. Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. `https://bitcoin.org/bitcoin.pdf`, 2009. Retrieved December 2016.

[75] Z. Németh, C. Pérez, and T. Priol. Workflow Enactment Based on a Chemical Methaphor. In *Proceedings of the 3rd IEEE International Conference on Software Engineering and Formal Methods*, Koblenz, Germany, Sept. 2005.

[76] R. D. Nicola, G. Ferrari, and R. Pugliese. KLAIM: A Kernel Language for Agents Interaction and Mobility. *IEEE Trans. Softw. Eng.*, 24(5):315–330, 1998.

[77] M. Obrovac. *Chemical Computing for Distributed Systems: Algorithms and Implementation.* PhD thesis, Université de Rennes 1, Mar. 2013.

[78] M. Obrovac and C. Tedeschi. Deployment and Evaluation of a Decentralised Runtime for Concurrent Rule-Based Programming Models. In *14th International Conference on Distributed Computing and Networking (ICDCN 2013)*, volume 7730, pages 408–422, Bombay, India, January, 3-6 2013.

[79] M. Obrovac and C. Tedeschi. Distributed Chemical Computing: A Feasibility Study. *International Journal of Unconventional Computing*, 9(3-4):203–236, 2013.

[80] M. Parashar and S. Hariri. Autonomic Computing: An Overview. In *International Workshop on Unconventional Programming Paradigms (UPP)*, pages 257–269, Le Mont Saint-Michel, France, Sept. 2004.

[81] G. Păun, G. Rozenberg, and A. Salomaa. *The Oxford Handbook of Membrane Computing.* Oxford University Press, Inc., New York, USA, 2010.

[82] M. J. Quinn. *Parallel Programming in C with MPI and OpenMP.* McGraw-Hill Education Group, 2003.

[83] Y. Radenac. *Programmation "chimique" d'ordre supérieur.* PhD thesis, Université de Rennes 1, April 2007.

[84] L. Ramakrishnan, S. S. Poon, V. Hendrix, D. K. Gunter, G. Z. Pastorello, and D. A. Agarwal. Experiences with User-Centered Design for the Tigres Workflow API. In *Proceedings of the 10th IEEE International Conference on e-Science*, pages 290–297. IEEE Computer Society, Oct. 2014.

[85] M. Raynal. A Distributed Solution to the k-out of-M Resources Allocation Problem. In *Advances in Computing and Information — ICCI'91*, volume 497 of *Lecture Notes in Computer Science*. Springer, 1991.

[86] J. Rojas Balderrama, M. Simonin, C. Morin, V. Hendrix, L. Ramakrishnan, D. Agarwal, and C. Tedeschi. Combining Workflow Templates with a Shared Space-based Execution Model. In *Proceedings of the 9th Workshop on Workflows in Support of Large-Scale Science*, New Orleans, LA, Nov. 2014.

[87] J. Rojas Balderrama, M. Simonin, and C. Tedeschi. GinFlow: A Decentralised Adaptive Workflow Execution Manager. In *30th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Chicago, USA, May 23-27 2016.

[88] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware'01)*, pages 329–350, London, UK, 2001. Springer-Verlag.

[89] C. Schmidt and M. Parashar. Squid: Enabling Search in DHT-based Systems. *J. Parallel Distrib. Comput.*, 68(7):962–975, 2008.

[90] F. B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: a Tutorial. *ACM Comput. Surv.*, 22, 1990.

[91] R. Sharma and A. Datta. Supernova: Super-Peers Based Architecture for Decentralized Online Social Networks. In *4th International Conference on Communication Systems and Networks (COMSNETS)*, pages 1–10. IEEE, 2012.

[92] J. Siegel. *CORBA 3 Fundamentals and Programming.* John Wiley & Sons, Inc., New York, USA, 2nd edition, 1999.

[93] D. Skeen and M. Stonebraker. A Formal Model of Crash Recovery in a Distributed System. *IEEE Transactions on Software Engineering*, SE-9(3), 1983.

[94] M. Sonntag, K. Gorlach, D. Karastoyanova, F. Leymann, and M. Reiter. Process Space-based Scientific Workflow Enactment. *International Journal of Business Process Integration and Management*, 5(1):32–44, 2010.

[95] V. Springel. The Cosmological Simulation Code GADGET-2. *Monthly Notices of the Royal Astronomical Society*, 364:1105–1134, 2005.

[96] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of the 2001 ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM'01)*, pages 149–160, 2001.

[97] Y. Suzuki and H. Tanaka. Symbolic Chemical System Based on Abstract Rewriting System and its Behavior Pattern. *Artificial Life and Robotics*, 1:211–219, 1997.

[98] C. Szyperski. *Component Software: Beyond Object-Oriented Programming - 2nd edition.* Addison-Wesley, 2002.

[99] R. Tolosana-Calasanz, J. A. Bañares, O. F. Rana, P. Álvarez, J. Ezpeleta, and A. Hoheisel. Adaptive Exception Handling for Scientific Workflows. *Concurrency Comput.: Pract. Exper.*, 22(5):617–642, 2010.

[100] A. M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society. Second Series*, 42:230–265, 1936.

[101] J. R. Ullmann. An Algorithm for Subgraph Isomorphism. *J. ACM*, 23(1):31–42, Jan. 1976.

[102] J. von Neumann. First Draft of a Report on the EDVAC. *IEEE Ann. Hist. Comput.*, 15(4):27–75, Oct. 1993.

[103] K. Wolstencroft, R. Haines, D. Fellows, A. Williams, D. Withers, S. Owen, S. Soiland-Reyes, I. Dunlop, A. Nenadic, P. Fisher, J. Bhagat, K. Belhajjame, F. Bacall, A. Hardisty, A. Nieva de la Hidalga, M. P. Balcazar Vargas, S. Sufi, and C. Goble. The Taverna Workflow Suite: Designing and Executing Workflows of Web Services on the Desktop, Web or in the Cloud. *Nucleic Acids Res.*, 41(Webserver-Issue):557–561, 2013.

[104] W. Yu. Consistent and Decentralized Orchestration of BPEL Processes. In *Proceedings of the 24th ACM Symposium on Applied Computing (SAC)*, Honolulu, Mar. 2009.

[105] R. Zivan and A. Meisels. Parallel Backtrack Search on DisCSP. In *Proceedings of the Workshop on Distributed Constraint Reasoning (DCR-02)*, Bologna, Italy, 2002.

[106] R. Zivan and A. Meisels. Message Delay and DisCSP Search Algorithms. *Annals of Mathematics and Artificial Intelligence*, 46(4):415–439, Apr. 2006.