HAL
archives-ouvertes.fr

Self-coordination of Workflow Execution Through Molecular Composition

Héctor Fernandez, Cédric Tedeschi, Thierry Priol

To cite this version:

Héctor Fernandez, Cédric Tedeschi, Thierry Priol. Self-coordination of Workflow Execution Through Molecular Composition. [Research Report] RR-7610, INRIA. 2011, pp.33. <inria-00590357v2>

HAL Id: inria-00590357

https://hal.inria.fr/inria-00590357v2

Submitted on 4 May 2011

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.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Self-coordination of Workflow Execution Through Molecular Composition

Héctor Fernandez — Cédric Tedeschi — Thierry Priol

N° 7610

April 2011

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--7610--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
RENNES - BRETAGNE ATLANTIQUE

Self-coordination of Workflow Execution Through Molecular Composition

Héctor Fernandez , Cédric Tedeschi, Thierry Priol

Theme :
Équipe-Projet Myriads

Rapport de recherche n° 7610 — April 2011 — 30 pages

Abstract: With the development of the Internet of Services, composing loosely-coupled, distributed and autonomous services dynamically has become one of the new challenges for large scale computing. While service composition systems are now a key feature of service oriented architectures, they are usually managed by a central coordination node, leading to performance and communication bottlenecks at runtime, as well as reliability issues. Accordingly, workflow executable languages, such as BPEL was designed to support centralized and static coordination of workflows. Thus, it appears crucial to promote service composition systems with a proper support for decentralized and dynamic coordination. Recently, nature metaphors have been shown of interest for inspiring autonomous coordination in service architectures. In this paper, we present a new analogy for service coordination based on *molecular composition*. Within this analogy, data and services are molecules floating and interacting freely in a chemical solution. The decentralized workflow execution coordination is achieved through a set of reactions between those molecules. In more concrete terms, we express this coordination with HOCL, a chemical higher-order language, in which reaction rules are themselves molecules able to react. These rules are composed, allowing a wide variety of workflow patterns to be executed. We here extend the notion of chemical computing, by proposing an executable chemical language for decentralized workflow execution.

Key-words: Service composition, Decentralization, Workflow execution, Nature-inspired models, Chemical programming paradigm

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex
Téléphone : +33 2 99 84 71 00 — Télécopie : +33 2 99 84 71 71

Auto-coordination de workflows par composition moléculaire

Résumé : Avec le développement de l'Internet des services, composer des services distribués faiblement couplés dynamiquement est devenu le nouveau *challenge* du calcul à large échelle. Alors que la composition de services est devenue un élément clef des plates-formes orientées service, les systèmes de composition de services suivent pour la plupart une approche centralisée, entraînant des problèmes à la fois techniques (goulot d'étranglement, tolérance aux pannes) mais aussi sociétaux ou environnementaux (protection de la vie privée, consommation d'énergie ...). Conjointement, des langages de description de *workflow*, tels BPEL, ne peuvent s'exécuter que dans des environnements statiques et centralisés. Il devient important de promouvoir des systèmes de composition de services permettant la coordination de ces services de façon décentralisée et autonome. Récemment, s'inspirer des processus naturels s'est avéré une piste prometteuse pour la coordination de services autonomes. Dans cet article, nous nous appuyons sur une analogie inspirée par la nature basée sur la *composition moléculaire*. Selon cette analogie, les services sont des molécules qui flottent dans une solution chimique. La coordination de ces services est effectuée par un ensemble de réactions entre ces molécules exprimant l'exécution décentralisée d'un *workflow*. Nous montrons comment combiner des règles de réactions (les règles étant elles-même des molécules) pour traiter une large variété de schéma de *workflow*. Dans cette voie, nous proposons une extension de la notion de calcul chimique pour l'exécution décentralisée et dynamique de *workflow*.

Mots-clés : calcul formel, base de formules, protocole, différentiation automatique, génération de code, modélisation, lien symbolique/numérique, matrice structurée, résolution de systèmes polynomiaux

Contents

1 Introduction **3**

2 The Chemical Metaphor **4**
2.1 Chemical Programming Paradigm 4
2.2 An Architecture for Distributed *Chemical* Coordination 6

3 Workflow Definition **6**

4 Molecular Compositions **9**
4.1 Chemical Workflow Representation 9
4.2 Global Variable Chemical Definition 11
4.3 Chemical Rules for Iteration Strategies 12
4.4 Chemical Rules for Distributed Execution 13
4.5 Solving Workflow Patterns 15

5 Execution Example **22**

6 Related Works **27**

7 Conclusion **28**

1 Introduction

Loose coupling has been one of the requirements of the development of service oriented architectures (SOA) [17]. It is also one of the keys to their success. Internet of services has now emerged as a global computing platform gathering myriads of autonomous heterogeneous services such as storage space, computing power, or more often software components offered to the users through the web. SOAs are now a multipurpose paradigm, facilitating business as well as helping scientific investigations based on computer-intensive applications. In both fields, the combinations of services allow to build more complex applications known as *composite web services* which are a temporal composition of services represented by a *workflow*, describing the data and control dependencies between services. The execution of such compositions are today managed by a central orchestrator responsible for the coordination of all data and control flows between services. This centralization in such managers leads to various weaknesses such as poor scalability, availability, and reliability [2]. Developing decentralized environments for these workflow managers appears crucial to tackle these issues. Accordingly, current workflow executable languages used within these systems, such as BPEL [16] provides concepts and constructs made for static workflows executed on centralized architectures and cannot adequately describe dynamic and decentralized processes [7, 12]. Likewise, in the scientific area, applications have an increasing need for efficiency, manageability and productivity. For instance, new workflow languages of this area are required to provide features like implicit parallelism (dependencies being discovered by the compilers) [24], and data-driven coordination, as done in scientific workflow systems like Taverna with Scufl [20] or Pegasus with DAX [10].

Lastly, nature metaphors have been shown of high interest for developing new approaches for service coordination [23]. In this paper, we build an approach based on the molecular composition analogy for the decentralized execution of a wide variety of workflow structures, referred to as *patterns* [1]. In our analogy, reactions modeling interactions between services at runtime are molecules to be composed and distributed among services involved in the execution of a given pattern. Our approach is based on the *chemical programming paradigm*, which is a high-level execution model. In this model, a computation is seen as a set of reactions consuming some molecules floating and interacting freely within a kind of membrane also known as *chemical solution*, and producing new ones. Reactions take place in an implicitly parallel, autonomous, and decentralized manner. This model naturally expresses distributed coordination [6]. More specifically, we rely on the Higher-Order Chemical Language (HOCL) [4] to express our decentralized coordination. HOCL provides the higher order: reaction rules can apply on other reaction rules, programs dynamically modifying programs. Based on HOCL and a fully decentralized *chemical* architectural framework [11], we build the relevant HOCL rules, compose and distribute them for solving a wide range of workflow patterns. In concrete words, our contribution is the construction of a decentralized coordination executable language, able to manage a wide range of patterns where the coordination responsibilities are distributed between all the ChWSes participating in a pattern.

Section 2.1 presents the chemical programming paradigm in more detail and our architectural framework. Section 4 details our decentralized coordination model and language, and how it can solve a wide variety of workflow patterns. Section 5 illustrates our contribution by an example of coordination of a complex workflow. Section 6 presents our related works. Section 7 concludes.

2 The Chemical Metaphor

Nature analogies, and more specifically bio-chemical metaphors, have recently gained momentum in the building of programming models coping with the requirements of the Internet of Services [23]. Initially proposed to naturally express highly parallel programs, the chemical programming paradigm exhibits properties required in emerging service platforms.

2.1 Chemical Programming Paradigm

According to the chemical metaphor, molecules (data) float in a chemical solution, and react according to reaction rules (program) producing new molecules (resulting data). These reactions take place in an implicitly parallel, autonomous, and non-deterministic way until no more reactions are possible, a state referred to as *inertia*. The computation is carried out according to local conditions without any central coordination, ordering or serialization. Nevertheless, this programming style can express both control and data driven executions, as we will detail in Section 4.4.

According to the early GAMMA formalization [5], the solution is a multiset containing the molecules, and reactions between molecules are rules rewriting the multiset. The multiset is the unique data structure natively supported by chemical programs. More recently, a higher-order chemical programming lan-

guage, called HOCL (*Higher Order Chemical Language*) [4], has been proposed. In HOCL, every entity is a molecule, including reaction rules. A program is a solution of molecules, formally a multiset of atoms, denoted A_1, A_2, ... ,A_n, ",," being the associative and commutative operator of construction of compound molecules. Atoms can be constants (integers, booleans, *etc.*), reaction rules, sub-solutions, denoted $\langle M_i \rangle$, or tuples, denoted A_1:A_2:...:A_n themselves composed of n atoms. A reaction involves a reaction rule **one** P **by** M **if** V and a molecule N satisfying the pattern P and the reaction condition V. The reaction consumes the rule and the molecule N, to produce a new molecule M. The basic **one** P **by** M **if** V reaction rule is one-shot: it is consumed when it reacts. Its variant **replace** P **by** M **if** V is n-shots: it is not consumed by reactions and stays within the solution for further reactions. For instance, consider the following program which calculates the maximum value of a given set of numbers.

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

The rule called max reacts with two integers x and y such that $x \geq y$ and replaces them by x alone. The program is a multiset containing max, and the integers. In order to extract the result from the solution, we introduce a higher-order rule responsible for the deletion of the max rule once the solution contains only the highest integer value and max. This introduces the need for sequentiality of events: we need to wait that all possible reactions between max and the couples of integers took place before deleting the rule. Within the chemical model, the sequentiality is achieved through sub-solutions: by definition, to react with molecules within a sub-solution, a reaction rule must wait for its inertia. In our example, this leads to the following encapsulation:

$$\langle \langle 2, 3, 5, 8, 9, max \rangle, \ \textbf{one } \langle max = m, \omega \rangle \textbf{ by } \omega \rangle$$

The m variable matches a rule named max, and ω is a particular notation that matches all the remaining elements. Initially, several reactions are possible: max can react with any couple of integers satisfying the condition. One possible execution scenario within the sub-solution is the following (2 and 8, as well as 3 and 5, react first, producing the intermediate state):

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

Once the inertia is reached within the sub-solution, the one-shot rule can be triggered, extracting the result:

$$\langle \langle 9, max \rangle, \ \textbf{one } \langle max = m, \omega \rangle \textbf{ by } \omega \rangle \rightarrow \langle 9 \rangle$$

This higher-order property gives a high expressiveness to HOCL, making of it a good candidate for expressing dynamic and autonomic coordination.

2.2 An Architecture for Distributed *Chemical* Coordination

We have proposed in [11] an architectural framework for a decentralized workflow coordination following a chemical approach. We now briefly review this architecture, illustrated on Figure 1.

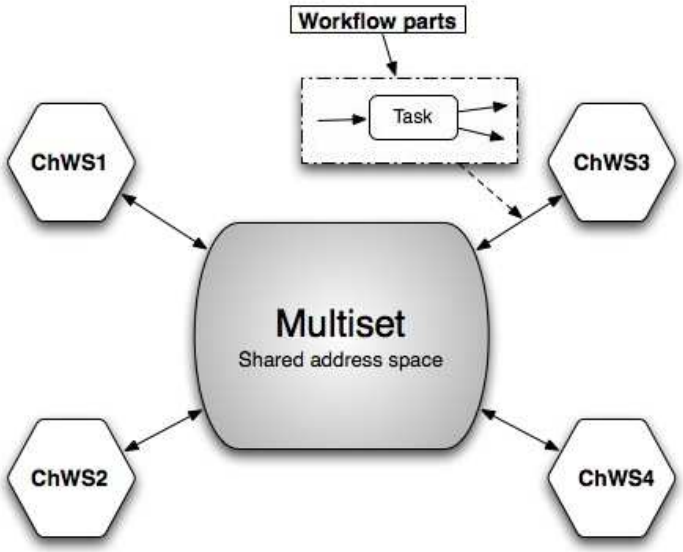

Figure 1: The proposed architecture.

This architecture is meant to allow several services to coordinate themselves following the chemical model. Their data and control dependencies, *i.e.*, their workflow, is represented as a multiset containing all the molecules of information (data and control flow) needed to describe and execute the workflow. However, this execution takes place inside each service that reads the information that concerns itself from the chemical solution into the multiset and (re)writing it at the end of the local execution. In these distributed settings, this multiset is now a space shared by all services.

To implement such a coordination, services involved are encapsulated in a *Chemical Web Service* (ChWS), integrating an HOCL interpreter playing the role of a local workflow (co-)engine. Physically, ChWSes are hosted by some nodes, themselves interconnected by a network; and logically identified by symbolic names inside the multiset.

3 Workflow Definition

Let us consider a workflow whose data and control dependencies of the workflow were previously defined at build-time using a traditional workflow definition language, such as the well-known BPEL (Business Process Executable Language)

	BPEL	**Scufl/Taverna**	**HOCL**
Web service definition	Defined using <*partnerLinks*>	Defined using <*processor*> plug-ins	Defined using *ChWS*:<...> molecules
Activity definition	Basic and structure activities	Data processing units	Chemical rules
Data definition	Explicit using variables	Implicit (input/output in data units)	Implicit or explicit using molecules
Semantic links	Transfer of control	Transfer of data	Transfer of data & control
Supported patterns	Sequence, parallel split, exclusive-choice, synchronization, simple-merge...	Sequence, parallel split and conditional	Sequence, parallel split, synchronization, simple-merge, exclusive-choice...
Parallel execution	Explicitly defined using <*flow*> primitive	Implicit	Implicit
Workflow management	Single coordinator node / Centralized	Centralized	Several coordinator nodes / Decentralized

Table 1: Comparison of BPEL, Scufl and HOCL

[16]. However, any workflow definition language could be used for translating one graphical workflow representation into a executable program. We now review several workflow languages and give the equivalences between these languages and an HOCL-based workflow definition that will be executed by HOCL engines.

BPEL is an imperative and control-based workflow language. It includes the explicit definition of the control flow that determines the order of execution. Likewise, in BPEL, the Web services are primitive execution blocks, and service composition is achieved using control primitives such as *sequence, parallel, conditionals* and *loops*. In contrast, workflow languages such as Scufl (language of the Taverna workflow management system) or HOCL are data-driven. Scufl is an XML-based workflow description language. Scufl defines an abstract workflow from a graph of data interactions between different services called *processors*, hiding the complexity of the interoperation of the services to the users. HOCL also presents a data-driven behavior, services are represented as chemical solutions, where data are represented as molecules and computations as the chemical reactions among molecules. HOCL can be also used as a hybrid language (both control and data driven) using some specific chemical rules. In particular, to provide this control-driven behavior, we need to define additional chemical rules, which are *generic*, i.e., independent of any workflow definition. These additional rules, that will be part of the HOCL workflow engine, allow to define the order of execution, as detailed in Section 4.5.

Among the variety of existing workflow languages in the business-oriented computing, such as XPDL, BPEL and YAWL, or Kepler, DAX and Scufl for scientific domain, we choose BPEL and Scufl the most representative of their domain. Thus, we present in more detail the differences between BPEL, HOCL and Taverna, summarized in the Table 1.

As we illustrated with this comparison between BPEL, HOCL and Taverna. Web services definitions are represented in BPEL using <*partnerLinks*> prim-

itive or in Taverna using <*processor*> primitive, while they are represented as ChWSes in HOCL, a ChWS represents one service participating in the workflow.

WS definitions. A BPEL process consists of steps, each step is an *activity*. Activities are a set of primitives like *invoke*, *reply*, *assign*, *flow* among others, which are used for common tasks. In Scufl, activities are data processing units with input/output ports that can be executed as soon as input data are received. Unlike in the chemical paradigm, we use chemical rules to execute these tasks, as summarized in Table 1.

Data definition. For data definition, in Table 1, BPEL requires the explicit definition of variables to hold data structures that are meant to be shared among activities. This definition takes additional effort but also brings more flexibility. For example, in BPEL you can define both *global variable* concerning the whole flow and *local variable* whose scope will be a specific activity. In HOCL, molecules within the main solution can be used as global variables without the need for explicit definition thanks to its data-driven behavior, as we will detail in Subsection 4.2. Likewise, BPEL variables of complex type must be initialized prior to their first use. However, this initialization is not required for the molecules in HOCL and neither in Taverna where the notion of data is directly linked from an output to an input with no initialization.

Data transfer. In function of the information transferred through links among activities or nodes, our system distributes control and data information about the execution. In the scientific workflow area, workflows take the shape of data processing pipelines requiring to express data transfer easily. That is the reason because Taverna is data-driven language. In contrast, in BPEL, control information is only transferred through links representing the order of execution.

Workflow management. In the classic orchestration model of BPEL, control dependencies and data are distributed through a centralized engine, which results in unnecessary data transfer, wasted bandwidth and the engine to become a bottleneck of the execution of workflows. In Taverna, although the language offer a distributed execution, its coordination is still managed by a centralized engine. In contrary, the chemical execution model is based on several chemical-local engines which are co-responsible of the coordination during the execution of workflows, as detailed in [11].

Workflow patterns support. Currently, most workflow languages support the basic construct of sequence, iterations, splits and joins. However, the interpretation of even these basic constructs is not uniform and it is often unclear how more complex workflow patterns could be supported. For instance, both BPEL and HOCL support a large set of workflow patterns such as sequence, synchronization or simple-merge. See [1] for more information on these constructs. These patterns are applied using some primitives in BPEL, and by the use of some specific chemical rules in HOCL. However, Taverna, because of its data-driven behavior only supports few workflow patterns such as sequence, conditional and parallel split, since the order of execution is specified by data-dependencies with no particular primitives.

Thus, we consider that any workflow definition can be translated into a chemical program thanks to the data-control driven coordination what we can provide to our chemical programs. HOCL integrates both the simplicity of data-driven control for simple patterns, while supporting complex workflow patterns by the definition of rules for control-driven dependancies. The next section illustrates how to represent a workflow using our chemical analogy for the service composition.

4 Molecular Compositions

Based on the architectural framework presented in Section 2.2, and leveraging the higher-order property of HOCL, we now focus on the expression of the autonomic and decentralized execution of a wide variety of workflow patterns by defining reaction rules, composing them, and distributing them over the set of services involved in a workflow. Our execution relies on the molecular vision of every entity involved in a workflow execution. In the following, molecules represent the ChWSes themselves, the data they process, their data and control dependencies, and the rules making the whole interact. It is then important to distinct two types of rules inside the multiset: (1) the rules describing data and control flow of a specific workflow to be executed, and (2) workflow-independent rules for the coordination of the execution of any workflow. The latter are referred to as *generic rules* in the following.

In Figure 2, an abstract workflow with several services is translated into a molecular composition. Each ChWS disposes of a library of available generic rules. Some of them will be used during the execution, depending on the patterns used in the workflow definition. As we will see in Section 5, this composition will react in chain at runtime, performing the execution. The following details the molecules used and their composition. First, Section 3 presents the chemical definition of a workflow. Then, Section 4.4 introduces the notion of generic rules allowing the decentralized workflow execution. Finally, the combination and distribution of molecules for solving various workflow patterns are given in Section 4.5.

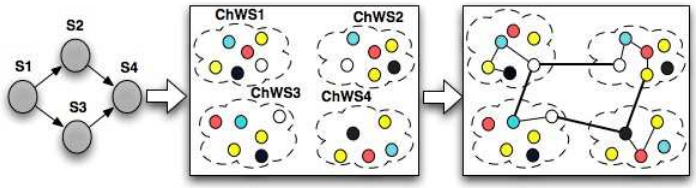

Figure 2: Molecular composition from an abstract workflow.

4.1 Chemical Workflow Representation

To express all data and control dependencies of a workflow, we use a series of chemical abstractions inspired by the work in [14]. The general shape of such a representation in Algorithm 3 is as follows: the main solution is composed of

as many sub-solutions as we have ChWSes in the workflow. Each sub-solution represents a ChWS with its data and control dependencies with other ChWSes. More formally, a ChWS is one molecule of the form $ChWSi : \langle \ldots \rangle$ where ChWSi refers to the symbolic name given to physical computational device that hosts the ChWSi and hidden its physical location.

```
1.01  ⟨ // Multiset (Solution)
1.02        ChWSi:⟨...⟩ // ChWS (Sub-solution)
1.03        ChWSi+1:⟨...⟩
1.04        ...
1.05        ChWSn:⟨...⟩
1.06  ⟩
```

Figure 3: General chemical workflow representation

Let us consider a simple workflow example illustrated by Figure 4. It is composed of four services S_1, S_2, S_3 and S_4. In this example, after S_1 completes, S_2 and S_3 can be invoked in parallel. Once S_2 and S_3 have completed, S_4 can be invoked.

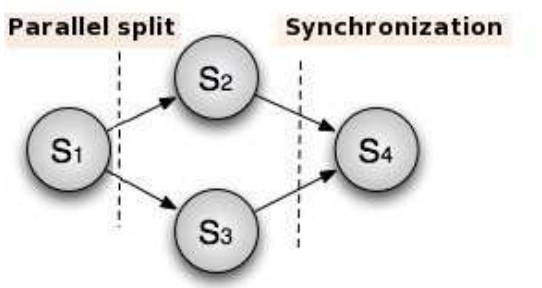

Figure 4: Simple workflow example.

The corresponding chemical representation for this workflow is presented in Figure 5. Remind that, the solution contains as many sub-solutions as services. $ChWS1 : \langle \ldots \rangle$ to $ChWS4 : \langle \ldots \rangle$ represent ChWSes. ChWSes are sub-solutions within the global solution. The links amongst ChWSes are expressed through a molecule of the form DEST:ChWSi with ChWSi being the destination ChWS where some information needs to be transferred. For instance, we can see in the ChWS1 sub-solution that ChWS1 must transfer some information (the result of ChWS1) to ChWS2 and ChWS3 (refer to line 2.01). Therefore, these links represent the distribution of information whose content is essential to coordinate the execution.

Let us focus on the details of the chemical representation of the workflow. As specified by this workflow, ChWS2 presents a data dependency, it requires a molecule RESULT*:ChWS1:value1* containing the result of S_1 to be performed (see the second part of line 2.02). The two molecules produced by the reaction represent the call to S_2 and their input parameters. They are expressed using a molecule of the form CALL*:Si*, and a molecule PARAM:$\langle in_1,\ldots,in_n \rangle$, where $in_1,\ldots,in_n$ represent the input parameters to call a service Si, In Figure 5 that

```
2.01  ChWS1:⟨Dest:ChWS2,Dest:ChWS3⟩,
2.02  ChWS2:⟨Dest:ChWS4, replace Result:ChWS1:value1 by Call:S2, Param:⟨(value1)⟩ ⟩,
2.03  ChWS3:⟨Dest:ChWS4, replace Result:ChWS1:value1 by Call:S3, Param:⟨(value1)⟩ ⟩,
2.04  ChWS4:⟨replace Result:ChWS2:value2, Result:ChWS3:value3
2.05            by Call:S4, Param:⟨(value2)⟩ ⟩
```

Figure 5: Chemical workflow representation.

input parameter corresponds with the result of some previous service S_j. ChWS3 works similarly.

In this representation, we show the control/data-driven behavior of our chemical language. Consider ChWS4. It needs to wait until ChWS2 and ChWS3 have been completed. This constitutes a control dependency known as *synchronization*. However, as we can see in line 2.05, the service S_4 is invoked only on *value2* which is the result of S_2. This constitutes a data dependency. The ChWS4 sub-solution contains one reaction rule translating those dependencies in chemical language (see line 2.05): the presence of molecules Result*:ChWS2:value2* and Result*:ChWS3:value3* inside the ChWS4 sub-solution expresses the fulfillment of the control dependencies. The input *value2* inside the Param:⟨ *value2* ⟩ molecule expresses the data dependency in ChWS4. During the execution, as soon as Result*:ChWS2:value2* and Result*:ChWS3:value3* appear in the ChWS4 sub-solution, the local engine of ChWS4 will be able to perform the reaction that will produce two molecules of the form Call*:S4* and Param:⟨ *value2* ⟩ to call the effective service S_4 on the input *value2*.

To sum up, one reaction rule can express both control and data dependencies. In contrast with the previous *synchronization* pattern, the simple data dependencies are enough to express the simple *parallel split* pattern of S_1 with S_2 and S_3. Thanks to the implicit parallelism of the chemical execution model, the reaction rules inside ChWS2 and ChWS3 can be executed in parallel. Therefore, ChWS2 and ChWS3 will receive the result of S_1 from ChWS1 and the invocation of S_2 and S_3 will take place in parallel.

This fragment of HOCL code is the chemical representation of a workflow, that will be interpreted by chemical local engines, performing the decentralized execution of this workflow thanks to a set of generic rules we introduce in next sections.

4.2 Global Variable Chemical Definition

Global variables in the context of workflows represents pieces of information that needs to be read multiple times by the different services involved. In chemical programming, this can be easily implemented through the notion of *multiplets*. Such a molecule can thus be consumed as many times as specified by its multiplicity. A multiplet consists in a specified number of identical molecules. For instance, 3^4 represents 4 instances of the molecule 3. In our context, a molecule m into a main solution of a workflow with a multiplicity n, such that m^n can be consumed in this workflow n times, n being virtually infinite.

4.3 Chemical Rules for Iteration Strategies

Most of scientific workflow management systems provide a set of *iteration strategies* by defining how input data received from other services are combined together for the computation. They specify how many times the service's task is invoked and what precise combination of input data is given to each of these invocations. *dot product* and *cross product* are the most common iteration strategies. We now detail how to support them in our chemical model.

Dot product. A dot product scheme produces tuples of data items with the same position in an arbitrary number of incoming branches of the service. The service is then launched once for each position, and produces an output located at the same position. The number of items in all input lists or arrays should be the same.

Chemical implementation. For the sake of readability, we here give the rules for a service with two incoming input branches to be composed with a dot product. This can be easily extended to an arbitrary number of incoming branches. A dot product involves a *dotProduct* rule where two molecules representing two lists of atoms are consumed to produce a unique molecule as output, DotProduct:⟨ ⟩. Each atom of this molecule corresponds with the other atoms located at the same position for each input list. For instance, as detailed in algorithm 1, the molecule produced by this dot product would be of the form DotProduct:⟨ *("a":1), ("b":2), ("c":3)* ⟩.

Algorithm 1 Dot product.

```
3.01  let dotProduct = replace List1:⟨ text, ω1 ⟩, List2:⟨ integer, ω2 ⟩, DotProduct:⟨ ω3 ⟩
3.02            by List1:⟨ ω1 ⟩, List2:⟨ ω2 ⟩, DotProduct:⟨ (text:integer), ω3 ⟩,
3.03  in
3.04  ⟨ dotProduct, List1:⟨"a","b","c"⟩, List2:⟨1,2,3⟩, DotProduct:⟨ ⟩ ⟩
```

Cross product. The cross product produces all possible data items combinations from an arbitrary number of incoming branches, each combination being made of one item of each incoming branch. The service task is then launched once for each if these combinations, and produces an output, indexed such that all indices of all inputs are concatenated into a multi-dimensional array.

Chemical implementation. A cross product involves four rules *crossProduct_start*, *crossProduct*, *crossProduct_list2End* and *crossProduct_end*. Again, we here detail the rules for two incoming branches. The *start_crossProduct* rule starts the execution by consuming two molecules representing two lists of atoms, producing two new molecules of the form CrossList1:⟨ ⟩ and CrossList2:⟨ ⟩ which will be used to calculate the cross product and one more molecule of the form CrossProduct:⟨ ⟩, where the temporary cross product result is stored. Then, the *crossProduct* rule iterates over all the atoms of the molecule CrossList2:⟨ ⟩ with the current first atom of the molecule CrossList1:⟨ ⟩. The *list2End* rule is in charge to iterate over all items of the molecule CrossList1:⟨ ⟩, and the *crossProductEnd* rule determines when all the atoms from the different lists have been consumed, introducing the final result into the solution.

Algorithm 2 Cross product.

```
4.01  let crossProduct_start = replace-one List1:⟨ text, ω1 ⟩, List2:⟨ integer, ω2 ⟩
4.02          by List1:⟨ ω1 ⟩, List2:⟨ ω2 ⟩, CrossProduct:⟨ ⟩, CrossList1:⟨ ω1 ⟩,
4.03          CrossList2:⟨ ω2 ⟩, crossProduct, list2End, crossProductEnd
4.04  let crossProduct = replace CrossList1:⟨ text, ω1 ⟩, CrossList2:⟨ integer, ω2 ⟩,
4.05              CrossProduct:⟨ ω3 ⟩
4.06      by CrossList1:⟨text, ω1 ⟩, CrossList2:⟨ ω2 ⟩, CrossProduct:⟨ (text,integer), ω3 ⟩,
4.07  let crossProduct_list2End = replace CrossList1:⟨ text, ω1 ⟩, CrossList2:⟨ ⟩, List2:⟨integer, ω2 ⟩
4.08          by CrossList1:⟨ ω1 ⟩, CrossList2:⟨ integer, ω2 ⟩, List2:⟨integer, ω2 ⟩
4.09  let crossProduct_end = replace CrossList1:⟨ ⟩, CrossList2:⟨ ⟩, CrossProduct:⟨ ω3 ⟩
4.10          by ω3
4.11  in
4.12  ⟨ crossProduct, List1:⟨"a","b","c","d"⟩, List2:⟨1,2,3,4⟩ ⟩
```

4.4 Chemical Rules for Distributed Execution

As previously mentioned, to ensure the execution of a chemical workflow, additional chemical *generic* rules (i.e., independent of any workflow) must be defined. In addition, for an efficient coordination, these rules use several specific molecules which represents the reactives and products generated during the reactions. Specific molecules allow to manage data related with the transfer of information, condition checking, faults detection and in applying the workflow patterns, in other words, information about the execution. By composing of these molecules, complex workflow patterns can be executed in a decentralized way among participants using chemical paradigm. How to distribute the workflow patterns responsibilities among participants is one of the common question that developers of decentralized workflow management systems take in account during the development of their systems. Next, we explain in detail some of these specific molecules, summarized in Table 2.

These molecules and rules are included in the chemical local engines and are responsible for the efficient execution of the workflow. We now review three of these *generic* rules, illustrated in Algorithm 3. First, we have rules in charge of the effective invocation of services: *invokeServ* and *preparePass*. The *invokeServ* rule invokes a Web service S_i, by consuming the tuples CALL*:Si* and PARAM*:*$\langle in_1,...,in_n \rangle$ representing the invocation to S_i and their input parameters inside the *ChWSi* sub-solution. The molecule FLAG_INVOKE*:1* is a flag where *value* indicates whether the invocation can take place. Thus, this execution triggers the call to service S_i (*i.e.*, the service associated with the ChWSi) and produces the result of the service invocation within the solution. In other words, such a rule constitutes an interface to the service invoked. The *preparePass* rule is used for preparing the transfer of these results to their destination, that will later trigger the execution of the *passInfo* rule.

Rule *passInfo* transfers molecules of information between ChWSes. This rule reacts with a molecule *ChWSj:*$\langle$PASS*:d:*$\langle\omega_1 \rangle\rangle$ that indicates that some molecules (here denoted ω_1) from ChWSj needs to be transfer to d. These molecules, once inside the sub-solution of d will trigger the next step of the execution. Therefore,

Molecules	Definition	Parameters
CALL:Si	Represent the service invocation.	Si: the url where wsdl file is located.
PARAM:⟨ in_1, ...,in_n ⟩		in_1,...,in_n: represents all the input parameters for a service invocation.
FLAG_INVOKE:value	Establish when the service invocation takes place.	value: 1 (start) | 0 (wait)
DISCRIMINATOR:value	Molecule used to activate an discriminator workflow pattern.	value: Yes | No
MERGE:value	Molecule used to activate an simple merge workflow pattern.	value: Yes | No
PASS:ChWSi:⟨ ω ⟩	Represent a molecule for the distribution of information.	ChWSi: destination chemical web service; ω: all molecules to be transferred
COND_PASS:value	Define the value of one condition.	value: 1 (true) | 0 (false)
COND_PASS:ChWSi:value	Define the value of one condition involving a ChWSi.	ChWSi: ChWS involved in this condition; value: 1 (true) | 0 (false)
ERROR:message	Contain un message of error.	message: information about an error
CANCEL:⟨ ω ⟩	Represent a molecule with the intercepted faults or error messages.	ω: contains messages about the error
CANCEL_CHWS:ChWSi	Define the chemical web service where the error information will be transferred.	ChWSi: destination chemical web service
DEST:ChWSi	Define the destination chemical web service for distribution of information.	ChWSi: destination chemical web service
SYNCMG_SRC:⟨ ChWSi, ω ⟩	Establish all the ChWS from which a molecule COMPLETED:ChWSi:⟨ ω ⟩ has to be received to start the execution of a destination ChWS. Used in Synchronization merge pattern.	(ChWSi, ω): incomming chemical web services
SYNC_SRC:⟨ ChWSi, ω ⟩	Rstablish all the ChWS from which a molecule COMPLETED:ChWSi:⟨ ω ⟩ has to be received to start the execution of a destination ChWS. Used in Synchronization pattern.	(ChWSi, ω): incomming chemical web services.
LOCKED:value	Establish when the execution of a reaction rule will be locked even whether it has all the required molecules.	value: 0 (unlocked)| 1 (locked)
RESET:⟨ ω ⟩	Represent a molecule whose content restarts to initial state of one particular solution as many times as it is necessary. Used in Multi merge pattern.	ω: all molecules to store into the solution
RESULT:ChWSi:⟨ ω ⟩	Contain the outcome of one service invocation for a ChWSi.	ChWSi: chemical web service already invoked; ω: contains the result of the invocation
COMPLETED:ChWSi:⟨ ω ⟩	Molecule representing one ChWSi whose execution have been completed.	ChWSi: chemical web service already invoked; ω: contains the result of the invocation
SYNCMG_INBOX:⟨COMPLETED:ChWSi:value, ω ⟩	Represent a molecule which contains all COMPLETED*:ChWSi:value* molecules already consumed. Used in combination with the molecule SYNCMG_SRC*:⟨ ChWSi, ω ⟩*	ω: represent the rest of molecules COMPLETED*:ChWSi:value* within the solution
SYNC_INBOX:⟨ COMPLETED:ChWSi:value, ω ⟩	Represent a molecule which contains all COMPLETED*:ChWSi:value* molecules already consumed. Used in combination with the molecule SYNC_SRC*:⟨ ChWSi, ω ⟩*	ω: represent the rest of molecules COMPLETED*:ChWSi:value* within the solution

Table 2: Specific molecules for the workflow execution

Algorithm 3 Basic generic rules.

```
5.01  let invokeServ = replace ChWSi:⟨Call:Si, Param:⟨in_1,...,in_n ⟩, Flag_Invoke:1, ω ⟩,
5.02                   by ChWSi:⟨Result:ChWSi:⟨value⟩ , ω ⟩
5.03  let preparePass = replace ChWSi:⟨Result:ChWSi:⟨value⟩ , Dest:ChWSj, ω ⟩
5.04                   by ChWSi:⟨Pass:ChWSj:⟨Completed:ChWSi:⟨value⟩ ⟩ ⟩
5.05  let passInfo = replace ChWSj:⟨Pass:ChWSi:⟨ ω_1 ⟩ , ω_2 ⟩, ChWSi:⟨ ω_3 ⟩
5.06                   by ChWSj:⟨ ω_2 ⟩, ChWSi:⟨ ω_1, ω_3 ⟩
```

the molecule ω_1 will be transferred from sub-solution *ChWSj* to sub-solution *ChWSi*, when reacting with *passInfo* rule.

These rules are the building blocks for decentralized execution. However, they can not, by themselves, solve how to *distribute the workflow patterns responsibilities among participants.*

4.5 Solving Workflow Patterns

A set of generic rules for solving complex workflow patterns are now presented, defining the control-logic of the execution. Note that, some of these rules involve every (source and destination) ChWSes participating in a pattern.

Parallel split pattern. A parallel split consists of one single thread splitting into multiple parallel threads (Figure 6.)

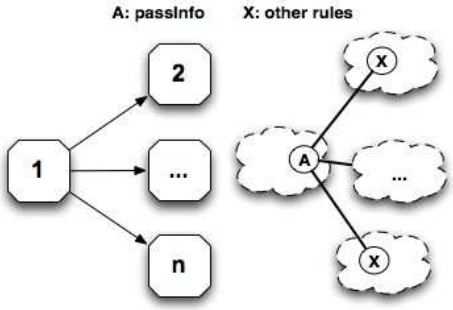

Figure 6: Parallel split.

Chemical representation: A parallel split pattern consists in the transfer of molecules produced on one (source) ChWS sub-solution to the others (destination). These reactions will be executed in parallel thanks to the implicit parallelism of the chemical model, so that all the information will be transferred in parallel to ChWSes. The *passInfo* rule has been explained in the algorithm 3.

Synchronization pattern. A Synchronization pattern is a process where multiple parallel branches converge to one single thread (Figure 7).

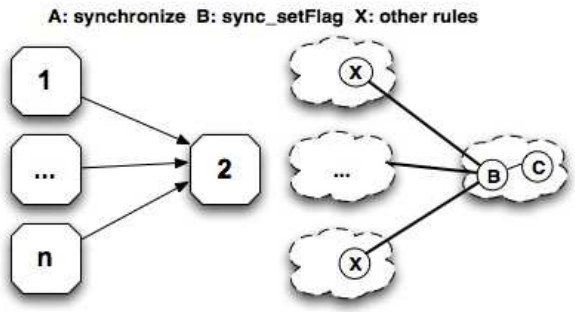

Figure 7: Synchronization.

Chemical implementation: A synchronization pattern involves two generic rules described in Algorithm 4. The *synchronize* rule allows to gather all the incoming COMPLETED*:ChWSi:⟨value⟩* molecules, specified by the molecule SYNC_SRC*:⟨ ChWSi,* ω_1 ⟩ representing all the ChWSes from which the destination ChWS needs to receive one molecule COMPLETED*:ChWSi:value* within its solution to trigger its own execution. When all molecules are gathered in the destination ChWS, another reaction, specified by the rule *sync_setFlag*, is triggered to produce a molecule FLAG_INVOKE*:1* allowing the service to be actually called through the *invokeServ* reaction (Line 6.04).

Algorithm 4 Chemical rules - Synchronization.

6.01 **let** *synchronize* = **replace** SYNC_SRC:⟨ChWSi, ω_1 ⟩, COMPLETED:ChWSi:⟨value⟩,
6.02 SYNC_INBOX:⟨ ω_2 ⟩
6.03 **by** SYNC_INBOX:⟨ COMPLETED:ChWSi:⟨value⟩, ω_2 ⟩, SYNC_SRC:⟨ ω_1 ⟩
6.04 **let** *sync_setFlag* = **replace-one** SYNC_SRC:⟨ ⟩ **by** FLAG_INVOKE:1

Molecular composition: synchronize (B) and *sync_setFlag* (C) rules are composed in the destination ChWS.

Exclusive choice pattern. An exclusive choice pattern selects one branch of the workflow among several, based on a decision process (Figure 8).

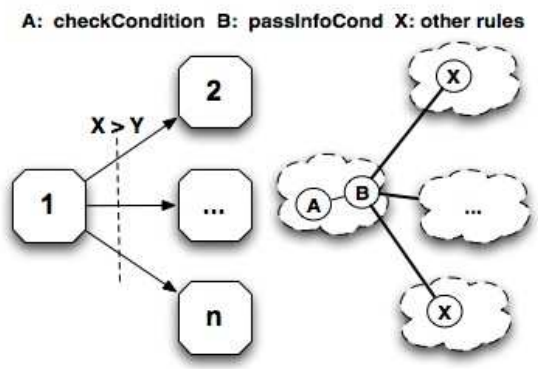

Figure 8: Exclusive choice.

Chemical implementation: An exclusive choice pattern involves the *passInfoCond* reaction rule, which is enabled when a given condition has been satisfied. This rule passes the information to the relevant destination ChWS, according to the satisfied condition. The molecule COND_PASS*:1* defines whether the condition has been satisfied. The multi-choice pattern, where one or several outgoing branches can be activated depending on a decision process, is also supported by the chemical engines in a similar way, as we will see later on this section.

Algorithm 5 Chemical rule - Exclusive Choice.

7.01 **let** *passInfoCond* = **replace** ChWSj:⟨PASS:ChWSi:⟨ ω_1 ⟩, COND_PASS:1, ω_2 ⟩, ChWSi:⟨ ω_3 ⟩
7.02 **by** ChWSi:⟨ ω_1, ω_3 ⟩, ChWSj:⟨ COND_PASS:1, ω_2 ⟩

Molecular composition: The *passInfoCond* rule (B) will be composed with the dynamic chemical rules in charge of checking the condition (A), transferring the molecule COND_PASS*:1* to the destination ChWSes.

Discriminator pattern. A Discriminator pattern is a structure in the workflow where a service will be activated by the first and only the first completed incoming branch. The remaining incoming branches will be ignored (Figure 9).

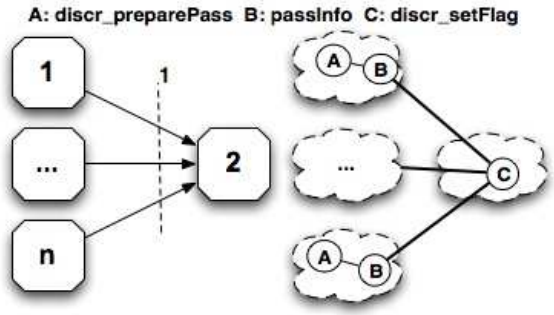

Figure 9: Discriminator.

Chemical implementation: As detailed in Algorithm 6, a discriminator pattern involves the *discr_ preparePass* reaction rule which, on every incoming branch, adds a DISCRIMINATOR*:Yes* molecule to the information to be passed into the destination service (Lines 8.01 and 8.02). The destination ChWS waits for this molecule and only the first DISCRIMINATOR*:Yes* molecule received will react. The FLAG_INVOKE*:1* molecule required to trigger the service invocation is created (Line 8.03). The following DISCRIMINATOR*:Yes* molecules received will be ignored.

Algorithm 6 Chemical rules - Discriminator.

8.01 **let** *discr_ preparePass* = **replace** DEST:ChWSj, RESULT:ChWSi:⟨value⟩
8.02 **by** PASS:ChWSj:⟨COMPLETED:ChWSi:⟨value⟩, DISCRIMINATOR:Yes⟩
8.03 **let** *discr_ setFlag* = **replace-one** DISCRIMINATOR:Yes **by** FLAG_INVOKE:1

Molecular composition: Each source ChWS has one *discr_ preparePass* (A) and one *passInfo* (B) rules, they are composed with *discr_ setFlag* rule (C) in the destination ChWS.

Simple merge pattern. A simple merge pattern describes the structure where two or more branches converge into a single service with no particular synchronization. The service will be launched only once. (See Figure 10).

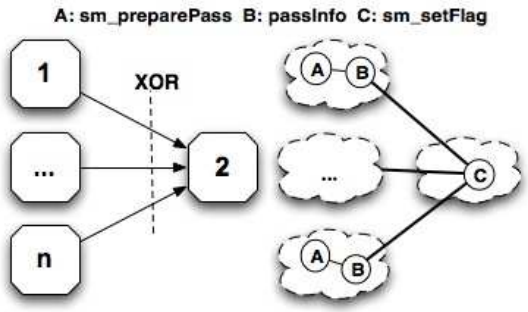

Figure 10: Simple merge.

Chemical implementation: A simple merge pattern involves the *sm_ preparePass* reaction rule which, on every source service, adds a MERGE*:Yes* molecule to the information to be passed into the destination service (see Lines 9.01 and 9.03). The destination ChWS waits for this molecule and only the first MERGE*:Yes* molecule received will be consumed. Next, *sm_ setFlag* reaction rule takes place producing one molecule of the form FLAG_INVOKE*:1*, allowing the service invocation. Likewise, the following MERGE*:Yes* molecules received will be ignored.

Algorithm 7 Chemical rules - Simple merge

9.01 **let** *sm_ preparePass* = **replace** DEST:ChWSj, RESULT:ChWSi:⟨value⟩
9.02 **by** PASS:ChWSj:⟨RESULT:ChWSi:⟨value⟩, MERGE:Yes⟩
9.03 **let** *sm_ setFlag* = **replace-one** MERGE:Yes **by** FLAG_INVOKE:1

Molecular composition: Each source ChWS has one *sm_ preparePass* (A) and one *passInfo* (B) rules, they are composed with *sm_ setFlag* rule (C) in the destination ChWS.

Synchronization merge pattern. The synchronization merge pattern allows to describe a service for which one or several of its incoming branches can be activated (through a previous multi-choice pattern). Then, the synchronization is required when several branches are active. Moreover, a branch that has already been activated, cannot be activated again while the merge is still waiting for other branches to complete (Figure 11).

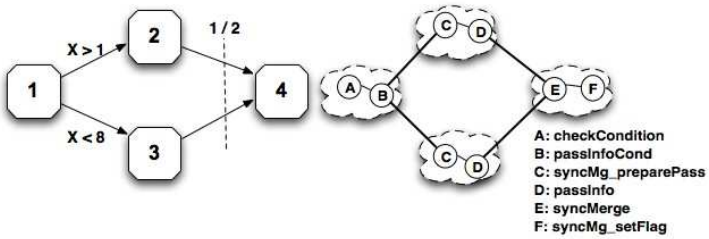

Figure 11: Synchronization merge.

Chemical implementation: As detailed in Algorithm 8, a synchronization merge pattern is achieved by transferring one molecule SyncMg_Src:⟨ *ChWSi,* ω ⟩ representing all the ChWSes from which one molecule of the form Completed:*ChWSi:*⟨*value*⟩ have to be received in the destination ChWS. Once, the destination ChWS contains all the needed molecules, it invokes its service. This molecule corresponds to all ChWSes whose branch to the destination ChWS has to be activated, and is generated by ChWS (by the molecule *A* on Figure 11). The multi-choice is then executed on service 1, actually activation of one or both services 2 and 3, through the *passInfoCond* rule. The *syncMerge* rule then waits for the required molecules and finally the *syncMg_setFlag* takes place producing a new molecule Flag_Invoke*:1*, allowing the invocation. The SyncMg_Inbox:⟨ ω ⟩ molecule stores the already received Completed:*ChWSi:*⟨*value*⟩ molecules.

Algorithm 8 Chemical rules - Synchronization merge.

```
10.01  let syncMg_preparePass = replace Dest:ChWSj, Result:ChWSi:⟨value⟩,
10.02                    SyncMg_Src:⟨ChWSi, ω ⟩
10.03                by Pass:ChWSj:⟨Completed:ChWSi:⟨value⟩, SyncMg_Src:⟨ChWSi, ω ⟩ ⟩,
10.04  let syncMerge = replace SyncMg_Src:⟨ ChWSi, ω1 ⟩ , Completed:ChWSi:⟨value⟩,
10.05                    SyncMg_Inbox:⟨ ω2 ⟩
10.06                by SyncMg_Inbox:⟨Completed:ChWSi:⟨value⟩, ω2 ⟩, SyncMg_Src:⟨ ω1 ⟩
10.07  let syncMg_setFlag = replace-one SyncMg_Src:⟨ ⟩ by Flag_Invoke:1
```

Molecular composition: The ChWS initiating the multi-choice includes a rule to decide on the condition, which will be used by the *passInfoCond* (*B*) to activate one or several of its outgoing branches. Then, each intermediate ChWS (encapsulating services 2 and 3) has a *syncMg_preparePass* rule (*C*), composed with *syncMerge* (*E*) and this with *syncMg_setFlag* rule (*F*) in the destination ChWS (encapsulating service 4, on which the merge should be achieved).

Multi merge pattern. A multi merge pattern describes the structure where two or more alternative branches converge again without synchronization into a single subsequent branch such that each enablement of an incoming branch will activate that subsequent branch. In particular, after a multi-choice pattern that can lead to several execution scenarios, multi-merge will, whatever the number of threads triggered by the multi-choice is, merge the threads into a single one.

Note that this workflow pattern is not supported by several engines based on the languages such as BPEL or XPDL, as explained in [18].

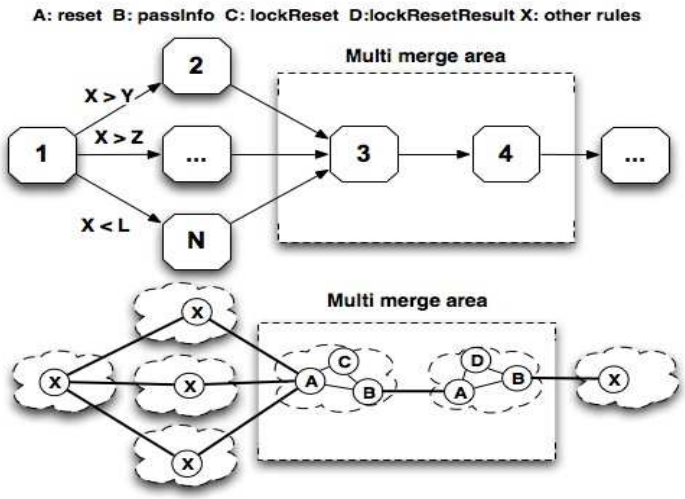

Figure 12: Multi merge.

Chemical implementation: As detailed in Algorithm 9, a multi merge pattern involves a set of reaction rules for restarting the initial state of several ChWSes involved in a *multi merge area*. The *mm_reset* rule consumes a molecule of the form Reset:⟨ ω ⟩ containing the required information to restart the initial state of one ChWS (all the molecules within its solution before the execution). Once the execution of a given incoming branch has finished and its result has been successfully transferred (denoted through a molecule of the form Success_Pass:*ChWSi*), the *mm_lockReset* rule reacts and produces all required molecules to restart the processing of a new incoming request. In particular, all ChWSes involved in this pattern, except the last ChWS (exit of the *multi merge area*, *node 4* in Figure 12) include the *mm_reset* and *mm_lockReset* rules (see Lines 11.01 to 11.04). For the last ChWS (*node 4*), the *mm_reset* and *lockReset_End* rules are used to return to the initial state and connect the merge area with the remaining of the workflow (see Lines 11.01 to 11.06).

Algorithm 9 Chemical rules - Multi merge.

11.01 **let** *mm_reset* = **replace-one** Reset:⟨ ω ⟩, Locked:0, Flag_Invoke:1
11.02 **by** ω, Reset:⟨ ω ⟩, Locked:1, Flag_Invoke:1
11.03 **let** *mm_lockReset* = **replace** Result:ChWSi:⟨value⟩, Locked:1, Success_Pass:ChWSi
11.04 **by** reset, Locked:0
11.05 **let** *mm_lockReset_End* = **replace** Result:ChWSi:⟨value⟩, Locked:1
11.06 **by** reset, Locked:0

Molecular composition: All ChWSes participating in this pattern have one *reset* (A) and *passInfo* (B) rules that will be composed with *lockReset_ End* rule (D) whether this ChWS represents the exit of our *multi merge area*, otherwise it will be composed with *lockReset* rule (C).

Cancel activity pattern. A cancel activity pattern describes an enabled task which is withdrawn prior to its execution. If the task has started, it is disabled and, where possible, the currently running instance is halted and removed (Figure 13).

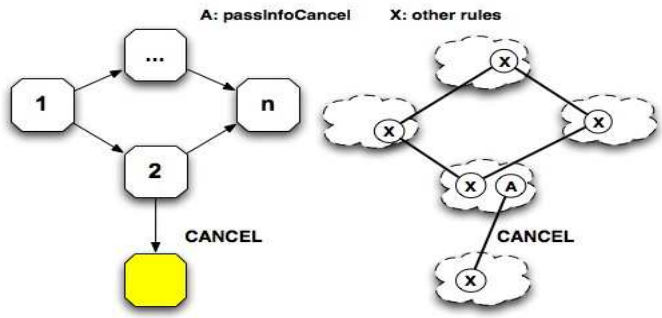

Figure 13: Cancel activity.

Chemical implementation: As detailed in Algorithm 10, a cancel activity pattern consists in the transfer of molecules containing error messages produced within the solution of one ChWS, to another ChWS called *CANCEL ChWS*. This *CANCEL ChWS* manages this information and halts the execution. A molecule of the form CANCEL:$\langle\ \omega\ \rangle$ contains the error messages. Similarly, the CANCEL_CHWS:*ChWSj* molecule contains the symbolic name of the *CANCEL ChWS*.

Algorithm 10 Chemical rules - Cancel activity.

12.01 **let** *passInfoCancel* = **replace-one** CANCEL:$\langle\ \omega_1\ \rangle$, CANCEL_CHWS:ChWSk, ChWSk:$\langle\ \omega_2\ \rangle$
12.02 **by** ChWSk:$\langle\ \omega_1, \omega_2\ \rangle$

Molecular composition: To apply this pattern one *passInfoCancel* (A) rule is composed with others rules with the aim of achieving to handle an error.

Thanks to these reaction rules, the coordination responsibilities can be distributed among all the ChWSes participating in a workflow pattern. By considering our chemical coordination model, the remaining workflow patterns identified by Van der Aalst et al. in [1] could be also handled.

5 Execution Example

In order to explain in detail the molecular composition (coordination between chemical engines), we present a workflow example, illustrated on Figure 14. This

figure shows on the top side seven ChWSes in applying *parallel split, synchronization, exclusive choice and discriminator* workflow patterns. On the bottom side, we shows the molecular composition graph representing that workflow. Following the execution, after *ChWS1* completes, it distributes the result to *ChWS2* and *ChWS3* in parallel. Once *ChWS2* and *ChWS3* have been completed, *ChWS4* can react. Next, *ChWS4* checks a condition and transfers some molecules to *ChWS5* and *ChWS6* whether it is satisfied. In that way, *ChWS5* and *ChWS6* are connected with *ChWS7* so that some information will be propagated to *ChWS7*. The *ChWS7* will reacts with the first received molecule from *ChWS5* or *ChWS6*, while the remaining molecules will be ignored. In a composition point of view, we show how each ChWS has a library of molecules where only some of them are composed for executing this workflow. This composition graph omits some molecules (data and reaction rules) from the library in each ChWS for space reasons.

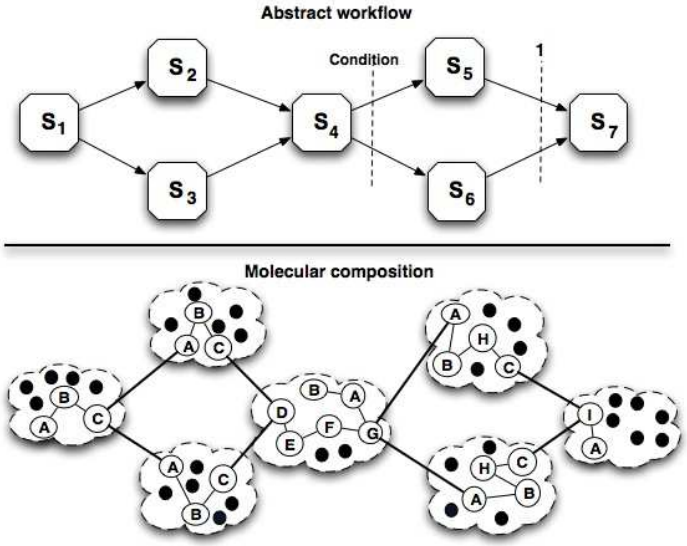

Figure 14: Example of Coordination.

Note that, thanks to the higher-order property, reaction rules react themselves with other molecules following the composition guidelines. The evolution of the HOCL representation of the workflow is given step by step in Figures 15 and 16. We refer to these three figures all along the section.

Let us first consider the block composed by *ChWS1, ChWS2, ChWS3* and *ChWS4* (Figure 15). *ChWS1* completed through *invokeServ* rule (molecule *A*), producing the result molecule RESULT*:ChWS1:⟨value⟩*. This molecule, through the *preparePass* rule (mol. *B*), is combined with the molecules DEST*:destination*, preparing the **parallel split**. *passInfo* (mol. *C*) triggers it by transferring in parallel the outcome of *ChWS1*.

Once the information is received by *ChWS2* and *ChWS3*, they launch the *invokeServ* rule (mol. *A*), producing two new molecules RESULT*:ChWS2:⟨value⟩* and RESULT*:ChWS3:⟨value⟩* that will be transferred similarly to *ChWS4* (Lines

13.25-14.05). Thus, *ChWS4* waits until the completion of *ChWS2* and *ChWS3*, thanks to the rules *synchronize* (mol. D) and *sync_ setFlag* (mol E).

Similarly, S_4 is invoked producing RESULT*:ChWS4:*⟨*value*⟩ (Line 14.10.) The ChWS4 then triggers the reaction rule (mol. F) in charge of checking the condition of the *exclusive choice* pattern and transfers some molecules to *ChWS5* or *ChWS6* according to the result, thanks to the *passInfoCond* rule (mol. G). *ChWS5* and *ChWS6* complete and produce their results (Lines 14.17 to 14.29). The *discr_ preparePass* rules (mol. H) are triggered by the engines of *ChWS5* and *ChWS6*. Two molecules PASS*:ChWS7:*⟨ COMPLETED*:ChWSi:*⟨*value*⟩*,* DISCRIMINATOR*:Yes* ⟩ are produced (Lines 14.23 to 14.25).

```
13.01 ChWS1:⟨Dest:ChWS2, Dest:ChWS3, passInfo, preparePass,invokeServ, * ⟩,
13.02 ChWS2:⟨Dest:ChWS4, passInfo, invokeServ, preparePass⟩,
13.03 ChWS3:⟨Dest:ChWS4, passInfo, invokeServ, preparePass⟩,
13.04 ChWS4:⟨Sync_Src:⟨ChWS2,ChWS3⟩, passInfoCond, synchronize, sync_setFlag, invokeServ,
13.05   preparePass, (replace-one condition_param by Cond_Pass:1, Dest:ChWS5,Dest:ChWS6
13.06             if ( condition )), Sync_Inbox:⟨ ω ⟩ ⟩,
13.07 ChWS5:⟨Dest:ChWS7, invokeServ, passInfo, discr_preparePass⟩,
13.08 ChWS6:⟨Dest:ChWS7, invokeServ , passInfo, discr_preparePass⟩,
13.09 ChWS7:⟨invokeServ, discr_setFlag⟩
                                ↓
13.10 ChWS1:⟨Dest:ChWS2, Dest:ChWS2, Result:ChWS1:⟨value⟩, passInfo, preparePass⟩,
13.11 ChWS2:⟨Dest:ChWS4, passInfo, preparePass, invokeServ⟩,
13.12 ChWS3:⟨Dest:ChWS4, passInfo, preparePass, invokeServ⟩,
13.13 ChWS4:⟨Sync_Src:⟨ChWS2,ChWS3⟩, passInfoCond, synchronize, sync_setFlag, invokeServ,
13.14   preparePass, (replace-one condition_param by Cond_Pass:1, Dest:ChWS5,Dest:ChWS6
13.15             if ( condition )), Sync_Inbox:⟨ ω ⟩ ⟩,
13.16 ...
                                ↓
13.17 ChWS1:⟨Pass:ChWS2:⟨Completed:ChWS1:⟨value⟩ ⟩,
13.18        passInfo, Pass:ChWS3:⟨Completed:ChWS1:⟨value⟩ ⟩ ⟩
13.19 ChWS2:⟨Dest:ChWS4, passInfo, preparePass, invokeServ⟩,
13.20 ChWS3:⟨Dest:ChWS4, passInfo, preparePass, invokeServ⟩,
13.21 ChWS4:⟨Sync_Src:⟨ChWS2,ChWS3⟩, passInfoCond, synchronize, sync_setFlag, invokeServ,
13.22   preparePass, (replace-one condition_param by Cond_Pass:1, Dest:ChWS5,Dest:ChWS6
13.23             if ( condition )), Sync_Inbox:⟨ ω ⟩ ⟩,
13.24 ...
                                ↓
13.25 ChWS1:⟨...⟩, ChWS2:⟨Dest:ChWS4, Result:ChWS2:⟨value⟩, passInfo, preparePass⟩,
13.26 ChWS3:⟨Dest:ChWS4, Result:ChWS3:⟨value⟩, passInfo, preparePass⟩,
13.27 ChWS4:⟨Sync_Src:⟨ChWS2,ChWS3⟩, passInfoCond, synchronize, sync_setFlag, invokeServ,
13.28   preparePass, (replace-one condition_param by Cond_Pass:1, Dest:ChWS5,Dest:ChWS6
13.29             if ( condition )), Sync_Inbox:⟨ ω ⟩ ⟩,
13.30 ...
                                ↓
13.31 ChWS1:⟨...⟩, ChWS2:⟨Pass:ChWS4:⟨Completed:ChWS2:⟨value⟩ ⟩, passInfo⟩,
13.32 ChWS3:⟨Pass:ChWS4:⟨Completed:ChWS3:⟨value⟩ ⟩, passInfo⟩,
13.33 ChWS4:⟨Sync_Src:⟨ChWS2,ChWS3⟩, passInfoCond, synchronize, sync_setFlag, invokeServ,
13.34   preparePass, (replace-one condition_param by Cond_Pass:1, Dest:ChWS5,Dest:ChWS6
13.35             if ( condition )), Sync_Inbox:⟨ ω ⟩ ⟩,
13.36 ...
```

Figure 15: Workflow execution, steps 0-4.

In the *ChWS5* and *ChWS6*, the *passInfo* rule (mol. C) propagates the molecule *Pass:ChWS7:⟨ information ⟩* to *ChWS7* (Lines 14.23-14.29). Once they are received by *ChWS7*, the *discr_setFlag* rule (mol. I) is consumed by the first *Discriminator:Yes* received, achieving the *discriminator* pattern. Also,

it triggers the *invokeServ* for the invocation of the S_7 producing the final result RESULT*:ChWS7:*⟨*value*⟩ (Line 14.32).

```
14.01 ChWS1:⟨...⟩,      ChWS2:⟨...⟩,      ChWS3:⟨...⟩,
14.02 ChWS4:⟨SYNC_SRC:⟨ ⟩, passInfoCond, synchronize, sync_setFlag, addResult,
14.03   SYNC_INBOX:⟨COMPLETED:ChWS2:⟨value⟩, COMPLETED:ChWS3:⟨value⟩ ⟩,
14.04   FLAG_INVOKE:1, (replace-one condition_param by COND_PASS:1,
14.05        DEST:ChWS5, DEST:ChWS6 if( condition )), invokeServ ⟩,
14.06 ...
                              ↓
14.07 ChWS1:⟨...⟩,      ChWS2:⟨...⟩,      ChWS3:⟨...⟩,
14.08 ChWS4:⟨SYNC_SRC:⟨ ⟩, passInfoCond, synchronize, sync_setFlag, preparePass
14.09   SYNC_INBOX:⟨COMPLETED:ChWS2:⟨value⟩, COMPLETED:ChWS3:⟨value⟩ ⟩,
14.10   RESULT:ChWS4:⟨value⟩, (replace-one condition_param by COND_PASS:1,
14.11                  DEST:ChWS5, DEST:ChWS6 if ( condition ))⟩,
14.12 ...
                              ↓
14.13 ChWS1:⟨...⟩,      ChWS2:⟨...⟩,      ChWS3:⟨...⟩,
14.14 ChWS4:⟨Pass:ChWS5:⟨COMPLETED:ChWS4:⟨value⟩ ⟩,passInfoCond
14.15   Pass:ChWS6:⟨COMPLETED:ChWS4:⟨value⟩ ⟩,...⟩,
14.16 ...
                              ↓
14.17 ChWS4:⟨...⟩,
14.18 ChWS5:⟨DEST:ChWS7, RESULT:ChWS4:⟨value⟩, invokeServ, passInfo,
14.19       discr_preparePass⟩,
14.20 ChWS6:⟨DEST:ChWS7, RESULT:ChWS4:⟨value⟩, invokeServ, passInfo,
14.21       discr_preparePass⟩,
14.22 ChWS7:⟨invokeServ, discr_setFlag⟩
                              ↓
14.23 ChWS4:⟨...⟩,
14.24 ChWS5:⟨Pass:ChWS7:⟨RESULT:ChWS5:⟨value⟩,DISCRIMINATOR:Yes⟩, passInfo⟩,
14.25 ChWS6:⟨Pass:ChWS7:⟨COMPLETED:ChWS6:⟨value⟩, DISCRIMINATOR:Yes⟩, passInfo⟩,
14.26 ChWS7:⟨invokeServ, discr_setFlag⟩
                              ↓
14.27 ChWS4:⟨...⟩,      ChWS5:⟨...⟩,
14.28 ChWS6:⟨Pass:ChWS7:⟨COMPLETED:ChWS6:⟨value⟩,DISCRIMINATOR:Yes⟩,passInfo⟩,
14.29 ChWS7:⟨invokeServ, RESULT:ChWS5:⟨value⟩, Discriminator:Yes, discr_setFlag⟩
                              ↓
14.30 ChWS4:⟨...⟩,      ChWS5:⟨...⟩,
14.31 ChWS6:⟨Pass:ChWS7:⟨COMPLETED:ChWS6:⟨value⟩, DISCRIMINATOR:Yes⟩, passInfo⟩,
14.32 ChWS7:⟨RESULT:ChWS7:⟨value⟩, ...⟩
```

Figure 16: Workflow execution, steps 5-11.

With this example, we have shown by composing of molecules from one library, a wide set of different workflows can be applied using our molecular composition analogy. Also we have shown that local engines within ChWSes are co-responsible for applying workflow patterns, invoking services, and prop-

agating the information to other ChWSes. The coordination is achieved as reactions become possible, in an asynchronous and decentralized manner.

6 Related Works

Earlier works [15], proposed distributed architecture based on Linda where distributed tuple spaces store into the tuples data and programs, allowing so the mobile computation by transferring and executing programs from one tuples to another. However, our approach provides a multiset where all the entities are considered molecules (data and programs "reaction rules") providing an easier way to match which data are needed to execute a chemical program on any ChWS. More recently, a series of works started to address the issue of decentralized workflow execution. The idea promoted in these works is to replace a centralized BPEL engine by a set of distributed, loosely coupled, cooperating nodes. To achieve this, one promising solution is to use a shared space as a coordination mechanism [9, 19], acting as a communication mean for exchanging control and data dependencies. Despite their architectural similarities with our approach, one key difference is that these approaches construct and execute a workflow in a low-level using BPEL or other proprietary languages. In particular, BPEL lacks of means to express dynamic behaviors and do not provide concepts for a distributed execution. Likewise, partitioning is a complex task, to be done statically at design time. In other words, there is not any simple algorithm to decentralize the execution of a composite web services using BPEL.

Some languages have then been proposed for providing that distributed support [21, 13]. However, these languages are finally turned into BPEL for the execution, losing some information in this translation. More related to scientific workflows, Scufl [20], DAX [10] or GWorkflowDL [3] workflow languages have been proposed for distributed environments. The Taverna system uses Scufl, an XML-based workflow description language to define an abstract workflow. DAX is an XML workflow description language used in the Pegasus system. Both offer implicit parallelism and are more simple and intuitive. However, they still offer a limited expressiveness in applying complex workflow patterns [3] due to its data-driven behavior. GWorkflowDL is a control/data driven language based on high-level Petri nets thus able to express more complex workflows. Some recent works try to use the BPEL standard by providing new extensions on the language or by designing a new architecture [8]. More generally, the vast majority of these approaches such as Taverna or Pegasus present architectures where the coordination is still highly centralized. In our approach, using a *chemical* inspired language brings a natural way to express dynamic behavior, both control and data driven coordination, implicit parallelism and distribution, and finally an easy way to construct and execute decentralized workflows.

Recently, works by Viroli *et al.* [22, 23] paved the way for new models of coordination inspired by nature. Our work is a concrete step forward in this way, focusing on decentralized workflow execution.

7 Conclusion

Although the design and implementation of new workflow management systems is a subject of considerable research, new solutions are still based on a centralized coordination model. Likewise, the workflow executable languages used in these systems such as BPEL are intrinsically static and do not provide concepts for distributed workflow execution. Other workflow languages using scientific purposes such as Scufl or DAX, even when executed in distributed environments, present a limited expressiveness, hindering them to support complex workflow patterns. Thus, it becomes crucial to promote a decentralized workflow execution systems whose executable language allows to partition a composite web service integrating complex patterns in fragments without losing information. We have proposed here a new analogy for service composition based on a chemical metaphor providing dynamic and autonomous behaviors. We have proposed the approach of *distributed molecular composition*. From this composition of molecules, a wide variety of workflow patterns as well as their decentralized and autonomous execution can be expressed. In that way, we are working on the implementation of one prototype in order to execute real workflows on our architecture and next to make comparative studies against current workflow management systems. In future, we also plan to evaluate in more detail the dynamic and autonomous features of our model using HOCL. Similarly, we want to express more information about service discovery and resources management using HOCL.

References

[1] Aalst, W.M.P.V.D., Hofstede, A.H.M.T., Kiepuszewski, B., Barros, A.P.: Workflow patterns. Distrib. Parallel Databases 14(1), 5–51 (2003), `http://portal.acm.org/citation.cfm?id=640516`

[2] Alonso, G., Mohan, C., Agrawal, D., Abbadi, A.E.: Functionality and limitations of current workflow management systems. IEEE Expert 12 (1997)

[3] Alt, M., Gorlatch, S., Hoheisel, A., werner Pohl, H.: A grid workflow language using high-level petri nets. In: Parallel processing and applied mathematics, 6th International Conference, PPAM 2005. vol. 3911, pp. 715–722. Springer Berlin, Pozna , Poland (2005)

[4] Banâtre, J., Fradet, P., Radenac, Y.: Generalised multisets for chemical programming. Mathematical Structures in Computer Science 16(4), 557–580 (2006)

[5] Banâtre, J.P., Métayer, D.L.: The gamma model and its discipline of programming. Sci. Comput. Program. 15(1), 55–77 (1990)

[6] Banâtre, J.P., Priol, T., Radenac, Y.: Chemical Programming of Future Service-oriented Architectures. Journal of Software 4(7), 738–746 (2009)

[7] Baresi, L., Nitto, E.D., Ghezzi, C., Guinea, S.: A framework for the deployment of adaptable web service compositions. SOCA pp. 75–91 (Apr 2007)

[8] Bosin, A., Dessí, N., Pes, B.: Extending the SOA paradigm to e-Science environments. Future Gener. Comput. Syst. 27(1), 20–31 (2011)

[9] Buhler, P.A., Vidal, J.M.: Enacting BPEL4WS specified workflows with multiagent systems. In Proceedings of the Workshop on Web Services and Agent-Based Engineering (2004)

[10] Deelman, E., Singh, G., Su, M., Blythe, J., Gil, Y., Kesselman, C., Mehta, G., Vahi, K., Berriman, G.B., Good, J., Laity, A., Jacob, J.C., Katz, D.S.: Pegasus: A framework for mapping complex scientific workflows onto distributed systems. Sci. Program. 13(3), 219–237 (2005)

[11] Fernández, H., Priol, T., Tedeschi, C.: Decentralized Approach for Execution of Composite Web Services using the Chemical Paradigm. In: 8th International Conference on Web Services (ICWS 2010). pp. 139–146. IEEE, Miami, USA (2010)

[12] Kunze, C., Zaplata, S., Lamersdorf, W.: Mobile process description and execution. In: International Conference on Distributed Applications and Interoperable Systems. pp. 32–47 (2006)

[13] Montagut, F., Molva, R.: Enabling pervasive execution of workflows. In: International Conference on Collaborative Computing: Networking, Applications and Worksharing. p. 10 pp. (2005)

[14] Németh, Z., Pérez, C., Priol, T.: Distributed workflow coordination: molecules and reactions. In: Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International (2006)

[15] Nicola, R.D., Ferrari, G., Pugliese, R.: KLAIM: a kernel language for agents interaction and mobility. IEEE TRANSACTIONS ON SOFTWARE ENGINEERING 24, 315—330 (1997), `http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.36.243`

[16] OASIS: Web services business process execution language,(WS-BPEL 2.0), (2007)

[17] Rosen, M., Lublinsky, B., Smith, K.T., Balcer, M.J.: Applied SOA: Service-Oriented Architecture and Design Strategies. Wiley (Jun 2008)

[18] Russell, N., Hofstede, A., van der Aalst, W., Mulyar, N.: Workflow Control-Flow patterns: A revised view. Tech. rep. (2006)

[19] Sonntag, M., Gorlach, K., Karastoyanova, D., Leymann, F., Reiter, M.: Process space-based scientific workflow enactment. International Journal of Business Process Integration and Management 5(1), 32 – 44 (2010)

[20] Tan, W., Missier, P., Madduri, R., Foster, I.: Building scientific workflow with taverna and BPEL: a comparative study in caGrid. In: ICSOC 2008 International Workshops, Sydney, Australia, pp. 118–129. Springer-Verlag (2009)

[21] Taylor, I.J., Deelman, E., Gannon, D.B., Shields, M., Slominski, A.: Adapting BPEL to scientific workflows. In: Workflows for e-Science, pp. 208–226. Springer (2007)

[22] Viroli, M., Casadei, M.: Biochemical tuple spaces for self-organising coordination. In: COORDINATION. pp. 143–162 (2009)

[23] Viroli, M., Zambonelli, F.: A biochemical approach to adaptive service ecosystems. Information Sciences pp. 1–17 (2009)

[24] Zhao, Y., Foster, I.: Scientific workflow systems for 21st century, new bottle or new wine. IEEE Workshop on Scienfitic Workflows (2008)

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399