Decentralized Workflow Coordination through Molecular Composition

Héctor Fernández[1], Cédric Tedeschi[2], and Thierry Priol[1]

[1] INRIA, France
[2] IRISA, University of Rennes 1 / INRIA, France
{hector.fernandez,cedric.tedeschi,thierry.priol}@inria.fr

Abstract. The dynamic composition of loosely-coupled, distributed and autonomous services is one of the new challenges of large scale computing. Hence, service composition systems are now a key feature of service-oriented architectures. However, such systems and associate languages strongly rely on centralized abstractions and runtime, what appears inadequate in the context of emerging platforms, like (federation) of clouds that can shrink or enlarge dynamically. It appears crucial to promote service composition systems with a proper support for autonomous, decentralized coordination of services over dynamic large-scale platforms. In this paper, we present an approach for the autonomous coordination of services involved in the execution of a workflow of services, relying on the analogy of molecular composition. In this scope, we trust in the chemical programming model, where programs are seen as molecules floating and interacting freely in a chemical solution. We build a library of molecules (data and reactions) written with HOCL, a higher-order chemical language, which, by composition, will allow a wide variety of workflow patterns to be executed. A proof of concept is given through the experimental results of the deployment of a software prototype implementing these concepts, showing their viability.

1 Introduction

Today, Information Technology (IT) applications tend more and more to be designed as a composition of services, following the wide adoption of service-oriented architecture (SOAs). These services are composed to form more complex services, commonly referred to as *workflows*. The problem of service composition gained recently a lot of attention from the industrial and academic research communities, both trying to define adequate service-based software engineering methods. Concomitantly, cloud computing has emerged as the next generation computing platform. One key property of clouds is *elasticity*, *i.e.*, the possibility for cloud users to dynamically extend the number of resources at their disposal.

Thus, future service composition systems should provide ways to express both workflows and platform characteristics. In particular, those systems must be able to deal with the high degree of parallelism and distribution of services over elastic platforms (especially in the case of federations of clouds). However,

G. Pallis et al. (Eds.): ICSOC 2011, LNCS 7221, pp. 22–32, 2012.

current service composition systems highly rely on a centralized vision of the coordination, failing to provide the right abstractions to express decentralization, and leading to various weaknesses at runtime such as poor scalability, availability, and reliability [3].

Lately, nature metaphors, and more specifically artificial chemistries [8], have been identified as a promising source of inspiration to express autonomous service coordination [20]. Among them, the *chemical programming paradigm* is a high-level execution model, where a computation is basically seen as a concurrent set of reactions consuming some molecules of data interacting freely within a *chemical solution* to produce new ones (resulting data). Reactions take place in an implicitly parallel, autonomous and decentralized manner. Recently, the Higher-Order Chemical Language (HOCL) [4] raised the chemical model to the higher-order, providing a highly-expressive paradigm: rules can apply on other rules, programs dynamically modifying programs, opening doors to dynamic adaptation and autonomous coordination [5].

Contribution. In this paper, we leverage the chemical analogy and extend it to *molecular composition* to model the decentralized execution of a wide variety of workflow structures, *a.k.a*, *workflow patterns* [19]. In our approach, services, as well as their control and data dependencies, are molecules interacting in the workflow's chemical solution. Chemical rules – higher-order molecules – are in charge of its decentralized execution, by triggering the required reactions locally and independently from each others. These local reactions, together, realize the execution of the specified workflow. A software prototype implementing these abstractions, has been developed, based on the HOCL chemical language. Experimentations over an actual platform connecting several geographically distributed clusters, have been conducted. For the sake of comparison, the prototype was developed in two versions – centralized and decentralized – and the performance gain obtained with the decentralized version over the centralized version are discussed in details.

Outline. Section 2 presents the background of our work: the chemical programming paradigm and its distributed architectural framework. Section 3 details our decentralized coordination model and language. Section 4 presents the software prototype, the experimental campaign and its results. Section 5 presents related works and Section 6 concludes the paper.

2 Chemical Computing

Bio-chemical analogies have been shown recently to be highly relevant in the design of autonomic service architectures [20]. In particular, the chemical programming paradigm exhibits the properties required in emerging cloud-based service architectures.

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). At runtime, reactions take place in an implicitly parallel and autonomous way, according to local conditions, without any central coordination, artificial ordering or serialization, until no more reactions are possible, a state referred to as *inertia*. This programming style allows to write programs keeping only the functional aspects of the problem to be solved.

In artificial chemistries [8], the solution is usually formally represented by a multiset of molecules, and reactions between molecules are rules rewriting this multiset. Recently, higher-order chemical programming has been proposed, through HOCL (*Higher-Order Chemical Language*) [4]. In HOCL, every entity is a molecule, including reaction rules. An HOCL program is basically a multiset of atoms, where atoms can be constants, tuples, sub-solutions, or reaction rules. A reaction involves a reaction rule **replace** P **by** M **if** V and a set of molecules denoted N satisfying the pattern P and the reaction condition V. The reaction consumes N, to produce a new molecule M. The alternative **replace-one** P **by** M **if** V can react only once, and is deleted in the reaction. Rules can be named. Let us consider the simple HOCL program below that extracts the maximum even number from a set of integers.

$$
\begin{array}{l}
\textbf{let } selectEvens = \ \textbf{replace } x, \omega \ \ \textbf{by } \omega \textbf{ if } x\%2 \ != 0 \\
\textbf{let } getMax = \ \textbf{replace } x, y \ \ \textbf{by } x \textbf{ if } x \geq y \\
\textbf{in } \langle \\
\quad \langle selectEvens, 2, 3, 5, 6, 8, 9 \rangle, \\
\quad \ \textbf{replace-one } \langle selectEvens = s, \omega \rangle \textbf{ by } getMax, \ \omega \\
\rangle
\end{array}
$$

The *selectEvens* rule removes odd numbers from the solution, by its repeated reactions with an integer x, ω denoting the whole solution in which *selectEvens* floats, deprived of x. The $getMax$ rule reacts with two integers x and y such that $x \geq y$ and replaces them by x. When triggered repeatedly, this rule extracts the maximum value from a solution of integers. The solution is composed by (i) a sub-solution containing the input integers along with the *selectEvens* rule, and (ii) a higher-order rule (third line of the solution) that *opens* the sub-solution, extracts the remaining (even) numbers and introduces the $getMax$ rule.

This program leverages the higher-order: a sub-solution can react with other molecules as soon as it has reached inertia itself. In other words, the $getMax$ rule reacts only after all odd numbers were removed from the solution.

$$\langle \ \langle selectEvens, 2, 6, 8 \rangle, \textbf{ replace-one } \langle selectEvens = s, \omega \rangle \textbf{ by } getMax, \ \omega \ \rangle$$

The higher-order rule then extracts remaining numbers, and introduces the $getMax$ rule, so triggering the second phase of the program where the maximum value is kept. The solution is:

$$\langle 2, 6, 8, getMax \rangle$$

$getMax$ then reacts with pairs of integers until only the value 8 remains. Note that, due to the higher-order, putting both rules directly in the integers solution

could result in a wrong result as the pipeline between the two rules would be broken, possibly leading to a wrong result, for instance if $getMax$ reacts first with the 8 and 9, thus deleting the value 8.

2.2 Distributed *Chemical* Coordination Architecture

The architectural framework used in this paper has been described in [9]. It allows several entities to coordinate with each others by reading and writing a shared multiset. It is illustrated in Figure 1. The workflow is represented by a multiset of molecules containing the information (data and control flow) needed to describe and execute it. The execution takes place inside each service reading the chemical solution from the multiset and (re)writing it at the end of each execution's step. In these distributed settings, this multiset is envisioned as a space shared by the services. More precisely, the multiset is composed of as many sub-solutions as there are ChWSes in the workflow, and is of the form:

$$\langle\ ChWS1 : \langle \ldots \rangle,\ ChWS2 : \langle \ldots \rangle,\ \ \ldots,\ ChWSn : \langle \ldots \rangle\ \rangle$$

Invocations of services are themselves encapsulated in a *Chemical Web Service* (ChWS), integrating an HOCL interpreter playing the role of a local workflow (co-)engine. Physically, ChWSes are running on some nodes of the targeted platform, and are logically identified by symbolic names into this multiset ($ChWSi$). In Figure 2, an abstract workflow with several services is translated into a molecular composition. Each ChWS has a library of available generic rules at its disposal. Some of these molecules are composed based on data molecule dependencies, molecules produced by one reaction triggerring other reactions, and so on.

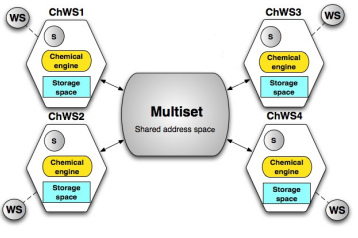

Fig. 1. The proposed architecture

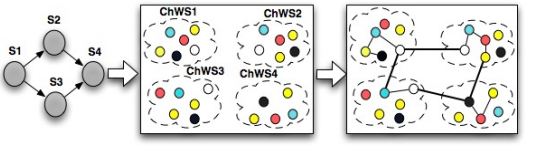

Fig. 2. Molecular composition from a workflow

3 Molecular Compositions

Based on the architectural framework presented in previous section, we now present the contribution of the paper: the expression of an autonomic and decentralized execution of various workflow patterns following an artificial molecular composition process. Molecules, reaction rules and their composition and distribution over services to compose, are presented. We distinguish two types of rules inside the multiset: (1) the rules describing data and control flows of a specific

workflow to be executed, and more importantly here, (2) workflow-independent rules for the coordination of the execution of any workflow referred to as *generic rules*. In the multiset, each sub-solution will only contain the molecules and rules (defined below) required to perform the desired patterns.

3.1 Generic Rules for Invocation and Transfer

Common tasks in a workflow of services are service invocation and information transfer between services. The three common rules responsible for these tasks are illustrated in Algorithm 1. They are present in virtually every existing pattern. The *invokeServ* rule, on reaction, invokes the actual Web Service S_i, by consuming the tuples CALL:*Si* whose presence indicates the potentiality of its invocation, and PARAM:$\langle\ in_1, \ldots, in_n\ \rangle$ representing its input parameters, and generates molecules encapsulating the results. The particular molecule FLAG_INVOKE, used in particular cases illustrated in the following, indicates that the invocation can actually take place. The *preparePass* rule is used to prepare messages containing the results to be transfered. Rule *passInfo*, on reaction, transfers the molecule ω_1 from sub-solution *ChWSi* to sub-solution *ChWSj*.

Algorithm 1. Common generic rules.

```
2.01 let invokeServ = replace ChWSi:⟨CALL:Si, PARAM:⟨in_1, ..., in_n⟩, FLAG_INVOKE, ω ⟩
2.02                  by ChWSi:⟨RESULT:ChWSi:⟨value⟩, ω ⟩
2.03 let preparePass = replace ChWSi:⟨RESULT:ChWSi:⟨value⟩, DEST:ChWSj, ω⟩
2.04                  by ChWSi:⟨PASS:ChWSj:⟨COMPLETED:ChWSi:⟨value⟩ ⟩, ω ⟩
2.05 let passInfo = replace ChWSi:⟨PASS:ChWSj:⟨ ω_1 ⟩, ω_2 ⟩, ChWSj:⟨ ω_3 ⟩
2.06                  by ChWSi:⟨ ω_2 ⟩, ChWSj:⟨ ω_1, ω_3 ⟩
```

3.2 Solving Workflow Patterns

We now present the details of the molecular compositions solving complex workflow patterns. A table of the details of all molecules used in the following with their details can be found in the research report [10].

Parallel Split Pattern. A parallel split consists of one single thread splitting into multiple parallel threads (See Figure 3.)
Chemical view: The data to be transferred, is one molecule produced inside the source ChWS sub-solution, and moved to the multiple destination ChWSes. Reactions for these transfers are triggered in parallel (implicitly, following the chemical model), following the *passInfo* rule (detailed before in Algorithm 1), and placed inside the sub-solution of the source ChWS (numbered 1 in Figure 3).

Synchronization Pattern. A Synchronization pattern is a process where multiple (source) parallel branches are merged into one single (destination) thread (See Figure 4).

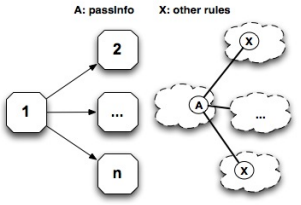

Fig. 3. Parallel split

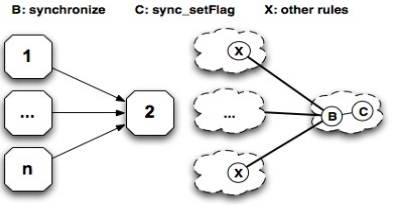

Fig. 4. Synchronization

Chemical view: A synchronization pattern should involve reactions able to gather all incoming result from the source ChWS sub-solutions on the destination ChWS. As detailed in Algorithm 2, the *synchronize* rule allows to gather all the incoming COMPLETED*:ChWSi:⟨value⟩* molecules, specified in the molecule SYNC_SRC*:⟨ ChWSi, ω_1 ⟩* specifying all sources. When all molecules are gathered on the destination ChWS, another reaction, specified by the rule *sync_setFlag*, is triggered to produce a molecule FLAG_INVOKE allowing the destination service (numbered 2 on Figure 4) to be called (Line 3.03).

Algorithm 2. Generic rules - Synchronization.

3.01 **let** *synchronize*=**replace** COMPLETED:ChWSi:⟨value⟩,SYNC_SRC:⟨ChWSi,ω_1 ⟩, SYNC_INBOX:⟨ ω_2⟩
3.02 **by** SYNC_INBOX:⟨ COMPLETED:ChWSi:⟨value⟩, ω_2 ⟩, SYNC_SRC:⟨ ω_1 ⟩
3.03 **let** *sync_setFlag* = **replace-one** SYNC_SRC:⟨ ⟩ **by** FLAG_INVOKE

Discriminator Pattern. In a discriminator pattern, a service is activated only once by its first completed incoming branch, ignoring any further completion of incoming branches. (See Figure 5).
Chemical view: Each source ChWS prepares a special message including a molecule indicating that discrimination is needed. As detailed in Algorithm 3, the *discr_preparePass* reaction rule, on every source ChWS, adds a DISCRIMINATOR molecule to the information sent (Lines 4.01 and 4.02). The destination ChWS waits for the DISCRIMINATOR molecule and only the first received will react with the one-shot *discr_setFlag* rule, setting the invocation flag (Line 4.03), and thus leading to the ignorance of subsequent incoming DISCRIMINATOR molecules.

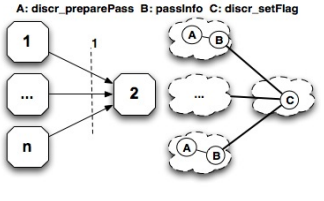

Fig. 5. Discriminator

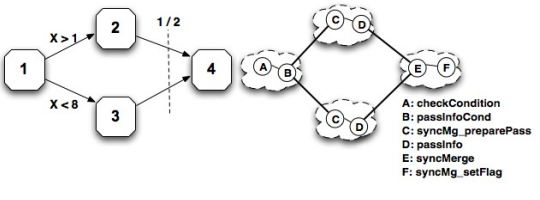

Fig. 6. Synchronization merge

Algorithm 3. Generic rules - Discriminator.

```
4.01 let discr_prepare Pass = replace DEST:ChWSj, RESULT:ChWSi:⟨value⟩
4.02                  by PASS:ChWSj:⟨ COMPLETED:ChWSi:⟨value⟩, DISCRIMINATOR ⟩
4.03 let discr_setFlag = replace-one DISCRIMINATOR by FLAG_INVOKE
```

Synchronization Merge Pattern. In a synchronization merge pattern, a service (service 4 on Figure 6) synchronizes several of its incoming branches. The number of branches that the service has to wait for is defined at runtime, depending on the result of a previous service (service 1 on Figure 6).
Chemical view: As detailed in Algorithm 4, the 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*⟩ has to be received in the destination ChWS. This molecule corresponds to all incoming branches that need to be activated. Following Figure 6, this is achieved by the molecule A in Service 1. The multi-choice is then executed on Service 1, actually activating 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* rule is triggered, producing a new molecule FLAG_INVOKE, allowing for the invocation. The SYNCMG_INBOX:⟨ ω ⟩ molecule stores the already received COMPLETED:*ChWSi:*⟨*value*⟩ molecules.

Algorithm 4. Generic rules - Synchronization merge.

```
5.01 let syncMg_prepare Pass = replace DEST:ChWSj, RESULT:ChWSi:⟨value⟩,
5.02                                   SYNCMG_SRC:⟨ChWSi, ω ⟩
5.03                  by PASS:ChWSj:⟨ COMPLETED:ChWSi:⟨value⟩, SYNCMG_SRC:⟨ ChWSi, ω ⟩ ⟩
5.04 let syncMerge = replace COMPLETED:ChWSi:⟨value⟩,
5.05                         SYNCMG_SRC:⟨ ChWSi, ω1 ⟩ , SYNCMG_INBOX:⟨ ω2 ⟩
5.06                  by SYNCMG_INBOX:⟨COMPLETED:ChWSi:⟨value⟩, ω2 ⟩, SYNCMG_SRC:⟨ ω1 ⟩
5.07 let syncMg_setFlag = replace-one SYNCMG_SRC:⟨ ⟩ by FLAG_INVOKE
```

To sum up, coordination responsibilities are distributed as reactions take place where molecules and rules are set. Other classical workflow patterns identified by Van der Aalst *et al.* in [19] such as *exclusive choice*, *multi-choice*, *simple merge*, *multi-merge*, or cancellation, can be treated similarly. We omit their description for space reasons. Their description can be found in the research report [10].

4 Experimental Evaluation

To establish a proof of concept on *molecular service composition*, we developed a software prototype, and deployed it over the nation-wide platform Grid'5000 [1], connecting together eight geographically-distributed federations of clusters.

4.1 Software Prototype

The prototype is written in Java, and relies on the *on-the-fly* compilation of HOCL programs. For the sake of comparison, and to highlight the advantages and drawbacks of our solution, two versions have been deployed. The first one, referred to as the *centralized* version, gathers all elements (service invocations and multiset) on a single physical machine. The second one, *decentralized*, deploys each ChWS, and the multiset on different machines, interconnected by the network. We now describe both versions in more details.

Centralized prototype. The centralized version presents a unique ChWS, a *chemical workflow service*, acting as a coordinator node, internally composed of a multiset playing the role of storage space and a workflow engine, as shown on Figure 7. Therefore, given a workflow definition, the unique ChWS executes the workflow by reading and writing the multiset. The interface between the chemical engine and the distant services themselves was implemented with the *service caller*, relying on the DAIOS framework [13] developed at Technische Universität Wien, which provides an abstraction layer allowing dynamic and transparent connections to different flavours of services (SOAP or RESTFul).

Decentralized prototype. The decentralized architecture differs in the sense that the multiset is now a shared space playing the role of a communication mechanism as well as a storage system. On each ChWS, a local storage space acts as a temporary container for its corresponding sub-solution to be processed by the local HOCL interpreter. As shown by Figure 8, ChWSes communicate with the multiset through the Java Message Service publisher/subscriber communication model. When the multiset detects changes in the sub-solution of a particular ChWS, it sends the new content to this ChWS. The sub-solution received is then processed, modified by the local HOCL interpreter and then published to the multiset for update. Each ChWS is now interfaced to one concrete WS, through the *service caller*.

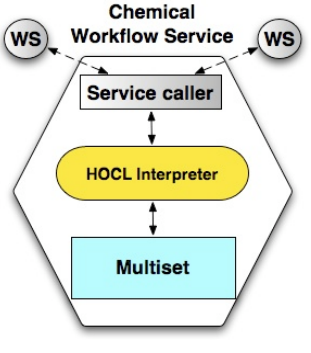

Fig. 7. Centralized prototype

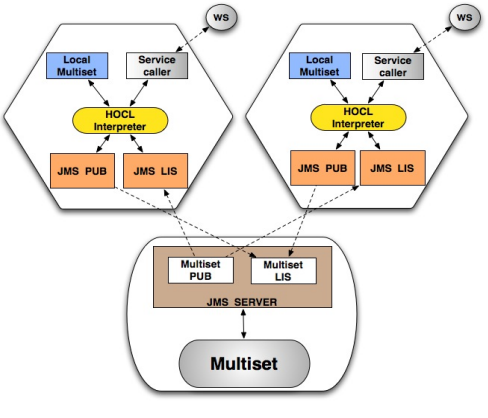

Fig. 8. Decentralized prototype

4.2 Results

We now present the performance of our approach with both propotype versions, using *synchronization*, *parallel split*, *discriminator* and *synchronization merge* patterns. These patterns can be extended in terms of number of services to obtain significant results regarding scalability.

Each pattern was deployed with 5, 15, 30, 45 and 60 tasks, where tasks are web services to be executed. Each node, in the decentralized prototype, is deployed on a distinct Grid'5000 node. Each experiment has been repeated 6 times. Charts show the results averaged on the 6 experiments. The *synchronisation* pattern consisted in one service realizing the synchronization, the others being its incoming branches. The *parallel split* consisted in one source node, the others being its outgoing branches. Similar *vertical* extensions have been done for the *discriminator* and *synchronization merge* patterns.

In Figure 9, performance obtained with the *synchronization* and *parallel split* patterns are given. A first observation is that decentralizing the process brings a non-negligible performance improvement, especially when the number of tasks to coordinate increases, the centralized version suffering from the concentrated workload on the unique coordinator for all the branches.

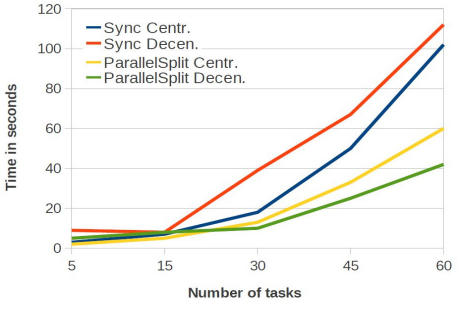

Fig. 9. Performance, basic patterns

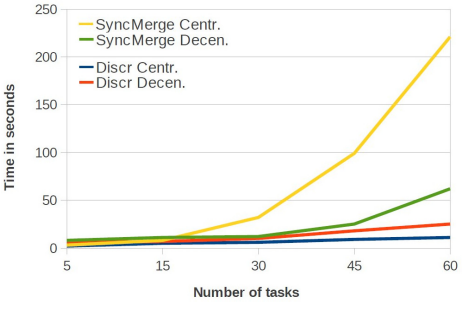

Fig. 10. Performance, advanced patterns

Next, we considered the more complex branching and merging concepts used in the *discriminator* and *synchronization merge* patterns. Results are shown in Figure 10. A first encouraging result is that the execution time for the *discriminator* shows similar performance evolution for both versions. The *synchronization merge* pattern highlights again the relevance of a decentralized approach, as the performance degradation in a centralized environment is similarly experienced. Again, this can be explained by the complexity of the pattern, composed of a *multi-choice* and a *synchronization* pattern, leading to a severe increase in the coordination's workload.

This series of experiments provides a proof of concept of the *chemical* service composition, and experimentally highlights the added value of a decentralized execution models to deal with the execution of large workflows.

5 Related Works

Early workflow executable languages, such as BPEL [16], YAWL [2], or other proprietary languages [14], lack of means to express dynamic behaviors. More recently, alternative approaches were proposed providing this dynamic nature to the service composition. A system governed and guided by rules mechanisms supporting dynamic binding and flexible service composition was proposed in [12]. In [7], a BPEL extension based on aspect oriented programming supports the dynamic adaptation of composition at runtime. Other approaches, such as [21], propose coordination models based on data-driven languages like Linda [11], to facilitate dynamic service matching and service composition. However, these approaches rely on architectures where the composition is still managed by a central coordinator node. In the same vein, works such as [15], propose a distributed architecture based on Linda where distributed tuple spaces store data and programs as tuples, allowing for mobile computations by transferring programs from one tuple to another. While our work presents similarities with this last approach, the chemical paradigm allows an increased abstraction level while providing support for dynamics.

A more recent series of works addresses the need for decentralization in workflow execution. The idea they promote is to replace a centralized BPEL engine by a set of distributed, loosely coupled, cooperating nodes [6, 17]. These works propose a system based on workflow management components: each workflow component contains sufficient information so that they can be managed by local nodes rather than one central coordinator. Again, this work present architectural similarities with our approach, but the chemical model allows for more expressive abstractions. Some languages have also been proposed for providing a distributed support to service coordination [18]. However, they are finally turned into BPEL for the execution, losing accuracy and expressiveness in the translation.

6 Conclusion

Although the design and implementation of workflow management systems is a subject of considerable research, most recent solutions are still mostly based on a centralized coordination model. Likewise, the workflow executable languages used in these systems are intrinsically static and do not provide concepts for autonomous and decentralized workflow execution. Thus, we need to promote a decentralized workflow execution systems, based on executable language allowing to partition composite web services integrating complex patterns into fragments without losing information.

In this paper, we have proposed concepts for a chemistry-inspired autonomic workflow execution, namely *molecular composition*. Such an analogy was shown to be able to express the decentralized and autonomous execution of a wide variety of workflow patterns. A ready-to-use HOCL library for this purpose has been proposed, and implemented in a middleware prototype. Its experimentation over a distribute platform has been conducted, providing a proof of concept and suggesting promising performance.

References

1. Grid'5000 (June 2011), http://www.grid5000.fr
2. van der Aalst, W.M.P., ter Hofstede, A.H.M.: YAWL: yet another workflow language. Information Systems 30(4), 245–275 (2005)
3. Alonso, G., Mohan, C., Agrawal, D., Abbadi, A.E.: Functionality and limitations of current workflow management systems. IEEE Expert 12 (1997)
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., Priol, T., Radenac, Y.: Chemical Programming of Future Service-oriented Architectures. Journal of Software 4(7), 738–746 (2009)
6. Buhler, P.A., Vidal, J.M.: Enacting BPEL4WS specified workflows with multiagent systems. In: Proceedings of the Workshop on WSABE (2004)
7. Charfi, A., Mezini, M.: AO4BPEL: an aspect-oriented extension to BPEL. World Wide Web 10, 309–344 (2007)
8. Dittrich, P., Ziegler, J., Banzhaf, W.: Artificial chemistries – a Review. Artificial Life 7, 225–275 (2001)
9. 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 (2010)
10. Fernández, H., Tedeschi, C., Priol, T.: Self-coordination of Workflow Execution Through Molecular Composition. Research Report RR-7610, INRIA (May 2011), http://hal.inria.fr/inria-00590357/PDF/RR-7610.pdf
11. Gelernter, D., Carriero, N.: Coordination languages and their significance. Commun. ACM 35(2), 96–107 (1992)
12. Laliwala, Z., Khosla, R., Majumdar, P., Chaudhary, S.: Semantic and rules based Event-Driven dynamic web services composition for automation of business processes. In: Services Computing Workshops, SCW 2006, pp. 175–182 (2006)
13. Leitner, P., Rosenberg, F., Dustdar, S.: Daios: Efficient dynamic web service invocation. IEEE Internet Computing 13(3), 72–80 (2009)
14. Montagut, F., Molva, R.: Enabling pervasive execution of workflows. In: International Conference on Collaborative Computing: Networking, Applications and Worksharing, p. 10 (2005)
15. Nicola, R.D., Ferrari, G., Pugliese, R.: KLAIM: a kernel language for agents interaction and mobility. IEEE Transactions On Software Engineering 24 (1997)
16. OASIS: Web services business process execution language, (WS-BPEL 2.0) (2007)
17. 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)
18. 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 (2007)
19. Van Der Aalst, W., Ter Hofstede, A., Kiepuszewski, B., Barros, A.: Workflow patterns. Distrib. Parallel Databases 14(1), 5–51 (2003)
20. Viroli, M., Zambonelli, F.: A biochemical approach to adaptive service ecosystems. Information Sciences, 1–17 (2009)
21. Wutke, D., Martin, D., Leymann, F.: Facilitating Complex Web Service Interactions through a Tuplespace Binding. In: Meier, R., Terzis, S. (eds.) DAIS 2008. LNCS, vol. 5053, pp. 275–280. Springer, Heidelberg (2008)