Chemical Programming of Future Service-oriented Architectures

Jean-Pierre Banâtre and Thierry Priol
INRIA
Campus de Beaulieu, F-35042 Rennes Cedex, France.
jean-pierre.banatre@inria.fr, thierry.priol@inria.fr

Abstract— **Service-based infrastructures are shaping tomorrow's distributed computing systems by allowing the design of loosely-coupled distributed applications based on the composition of services spread over a set of resources available on the Internet. Compared to previous approaches such as remote procedure call, distributed objects or components, this new paradigm makes feasible the loose coupling of software modules, encapsulated into services, by allowing a late binding to them at runtime. In this context, an important issue is how to express the composition of services while keeping this loosely-coupled property. Different approaches have been proposed to express services composition, mostly using specialized languages. This article presents and explore an unconventional new approach for service composition based on a programming language, inspired by a chemical metaphor, called the High-Order Chemical Language (HOCL). The proposed approach provides a very abstract and generic way of programming service composition thanks to the high-order property of HOCL. We illustrated this approach by applying it to a simple example that aims at providing a travel organizer service based on the composition of several basic and smaller services.**

I. INTRODUCTION

Service-based infrastructures are shaping tomorrow's distributed computing systems. It is rather difficult to come up with a strict definition, widely accepted, of what is a service. In the scope of this paper, a service can be seen as a set of functionalities that are available on a machine or through a network. After several attempts to design distributed programming paradigms, such as remote-procedure call [1], distributed objects [2] or distributed components [3], the service paradigm seems to solve one of the main issue when dealing with distributed systems: how to design loosely-coupled distributed applications based on the composition of a set of independent software modules called services that are spread over a set of resources available on a network such as the Internet. The loosely-coupled aspect is important when dealing with a distributed system. It allows an application to adopt a late binding to software modules. Services are discovered and brokered at runtime and bound when needed. This provides a lot of flexibility enabling the selection of the best services, in terms of Qualities of Service (QoS) such as performance and cost, but also to cope with failures since a given service can be replaced at runtime. It is foreseen in the near future that the programming of distributed applications will be just the expression of the composition of services available on the Internet. In fact, the Internet will be considered as a large scale computing system that shares some similarities with the microprocessors we have in our desktop or laptop computers but of course with a larger computing granularity. Internet will provide access, acting as a bus, to a large number of processing and storage units under the form of utility computing systems such as Grids [4] or Cloud computers like the Google [5] and Amazon [6] ones. To conclude with this analogy, services could be considered as the "instruction set" of such distributed computing infrastructures. With such analogy, the issue that is immediately emerging is how to express the instruction and data flows? Another issue that prevents us to go further in the analogy between a microprocessor and a service-based computing infrastructure is that failures can occur at any time and it is considered as a basic property of any distributed system.

Expressing the control and data flows, or simply workflows, in such large scale distributed computing infrastructures is thus challenging in many aspects. We think that existing approaches to express workflows need to be rethought to take into account the large scale dimension of these infrastructures allowing massively parallel coarse-grain computations and the dynamicity due to frequent failures. This paper proposes the "Chemical programming" approach as a candidate for expressing service coordination. This unconventional model of computing possesses two nice properties: it is implicitly parallel and autonomic. It gets its inspiration from the chemical metaphor, formally represented here by a chemical language called HOCL which stands for Higher-Order Chemical Language [7]. In HOCL, computation is seen as reactions between molecules in a chemical solution. HOCL is higher-order: reaction rules are molecules that can be manipulated like any other molecules, *i.e.*, HOCL programs can manipulate other HOCL programs. Reactions only occur locally between few molecules that are chosen non-deterministically. The execution is implicitly parallel since several reactions can occur simultaneously and it can also be seen as chaotic and possesses nice autonomic properties as shown in [8]. This model has

This paper is based on "Service Orchestration Using the Chemical Metaphor" by J-P. Banâtre, T. Priol and Y. Radenac which appeared in the Proceedings of the Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS 2008), Capri, Italy, October 2008. © 2008 Springer.

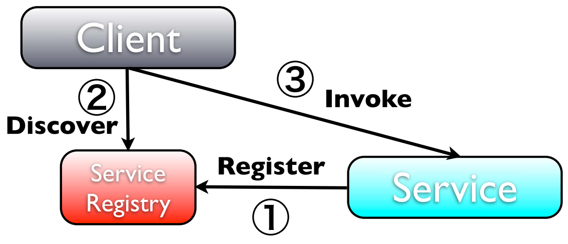

Figure 1. Service Oriented Computing

already been applied in the contexts of Grid workflow enactment [9], [10] and of Desktop Grids [11] and shown its suitability to express coordination of computations.

The objective of this paper is to show, through an example, that chemical programming can be a good candidate for service programming, such as the composition and coordination of services. On one side, applications are programmed in an abstract manner describing essentially the chemical coordination between (not necessarily chemical) abstract services. On the other side, chemical programs are specifically provided to the service run-time system in order to obtain the expected qualities of service in terms of efficiency, reliability, security, *etc.* These programs can be seen as special coordination programs providing guidelines to the runtime system allowing a better use of resources in order to obtain the expected Quality of Service.

The remaining of the paper is as follows. Section II gives an introduction to service oriented computing. Section III describes chemical programming model through the HOCL language. In section IV, we present some of the main existing approaches to express the composition of services in the Web Services framework and in particular those that share some similarities. Section V shows how HOCL can be used to express coordination but in a wider context and not only restricted to services. Section VI presents the proposed approach for service composition based on HOCL. Finally, we conclude in Section VII.

II. Service Oriented Computing

Service oriented computing is becoming a prominent programming style when dealing with large scale distributed systems made available by the Internet. It relies heavily on a set of standards to describe services, to discover them and finally to invoke them. Figure 1 shows how services can be invoked by a client. The first step is to discover a service through a look up phase that consists in calling (2) a service registry in which services have been previously registered (1). Once the registry has provided to the client the address of the service, the client can invoke it (3) through an interoperable communication protocol. This simple process is very similar to previous approaches such as distributed objects, like CORBA [2]. However, it differs by many characteristics. First of all, service oriented computing promotes late binding to services. A client can bind to a service just before its invocation. Moreover, a service is usually stateless, although there was an effort to support stateful services [12]. This stateless aspect of services is important to design loosely-coupled distributed applications and to promote the reusability of services. Context associated with stateless services is not stored in the service itself but is communicated through the exchange of messages between services whereas stateful services manage states internally. This state is modified depending on a sequence of calls to other services adding thus some dependencies and reducing both the reusability and the loose-coupling characteristics usually associated with service oriented computing.

Designing distributed applications using a service oriented computing approach requires that the designer identifies which services are required and how they are coordinated in time or in space. Orchestration and choreography are the two main composition paradigms for service composition. They are described later in the paper (see section IV).

On the technological side, service oriented computing is often linked with Web Services although it is not the only implementation of the service oriented computing concept. Web Service represents, however, the predominant available technology. The novel approach for service composition described in this paper is generic enough to be applied to various service oriented computing technologies.

III. Chemical Programming Model

A chemical program can be seen as a (symbolic) chemical solution where data is represented by “floating” molecules and computation by chemical reactions between them. When some molecules match and fulfill a reaction condition, they are replaced by the result of the reaction. That process goes on until an inert solution is reached: the solution is said to be inert when no reaction can occur anymore. Formally, a chemical solution is represented by a multiset and reaction rules specify multiset rewritings. There is an obvious mapping between programming objects known to the programmers and chemistry as shown in the following table:

Programming Object	**Chemistry**
Data	Molecule
Multiset	Solution
Computation	Reaction

The analogy between computing and chemisty applies likewise to the execution model for which properties of the execution model such as implicit parallelism and non determinism are represented by the brownian motion.

Properties	**Chemistry**
Implicit parallelism	Brownian
Non determinism	Motion

We use a higher-order chemical programming language called HOCL [7]. HOCL is based on the γ-calculus [13], a higher-order chemical computation model which can be seen as an higher-order extension of the Gamma language [14]. In HOCL, every entity is a molecule, including reaction rules.

A program is a molecule, that is to say, a multiset of atoms $(A_1, \ldots, A_n)$ which can be constants (integers, booleans, *etc.*), sub-solutions ($\langle M \rangle$) or reaction rules. Compound molecules (M_1, M_2) are built using the associative and commutative (AC) operator ",". The corresponding AC laws formalize the Brownian motion and can always be used to reorganize molecules.

The execution of a chemical program consists in applying reactions until the solution becomes inert.

A reaction involves a reaction rule $\mathbf{one}\, P\, \mathbf{by}\, M\, \mathbf{if}\, V$ and a molecule N that satisfies the pattern P and the reaction condition V. The reaction consumes the rule and the molecule N, and produces M. Formally:

$$\begin{aligned}(\mathbf{one}\, P\, \mathbf{by}\, M\, \mathbf{if}\, V),\ N &\longrightarrow \phi M \\ &\text{if } P \text{ match } N = \phi \text{ and } \phi V\end{aligned}$$

where ϕ is the substitution obtained by matching N with P. It maps every variable defined in P to a sub-molecule from N. For example, the rules in the following solution:

$$\begin{aligned}\langle 0,\ 10,\ 12,\ &\mathbf{one}\, x{::}\mathrm{Int}\, \mathbf{by}\, x+1\, \mathbf{if}\, x >= 9, \\ &\mathbf{one}\, x{::}\mathrm{Int}\, \mathbf{by}\, x-1\, \mathbf{if}\, x >= 10 \rangle\end{aligned}$$

produces several different results, due to non-determinism, according to which one-shot rule is applied first on the elements stored in the multiset. One of the following inert solution is produced by the execution: $\langle 0,\ 10,\ 12\rangle$, $\langle 0,\ 11,\ 11\rangle$ or $\langle 0,\ 9,\ 13\rangle$.

A molecule inside a solution cannot react with a molecule outside the solution (the construct $\langle . \rangle$ can be seen as a membrane). A HOCL program is a solution which can contain reaction rules that manipulate other molecules (reaction rules, sub-solutions, *etc.*) of the solution.

In the remaining of the paper, we use some syntactic sugar such as declarations $\mathbf{let}\, x = M_1\, \mathbf{in}\, M_2$ which is equivalent to M_2 where all the free occurrences of x are replaced by M_1. The reaction rules $\mathbf{one}\, P\, \mathbf{by}\, M\, \mathbf{if}\, C$ are one-shot: they are consumed when they react. Their recursive variant denoted by $\mathbf{replace}\, P\, \mathbf{by}\, M\, \mathbf{if}\, C$ are n-shot, *i.e.*, they do not disappear when they react (like in Gamma).

There are usually many possible reactions making the execution of chemical programs highly parallel and non-deterministic. Since reactions involve only a few molecules and react independently of the context, many distinct reactions can occur at the same time. For example, consider the following program that computes the prime numbers lower than 10 using a chemical version of the Eratosthenes' sieve:

$$\begin{aligned}&\mathbf{let}\, \mathsf{sieve} = \mathbf{replace}\, x, y\, \mathbf{by}\, x\, \mathbf{if}\, x \text{ div } y\, \mathbf{in} \\ &\langle \mathsf{sieve}, 2, 3, 4, 5, 6, 7, 8, 9, 10 \rangle\end{aligned}$$

The rule sieve reacts with two integers x and y such that x divides y, and replaces them by x (*i.e.*, removes y). Initially several reactions are possible, for example $\mathsf{sieve}, 2, 8$ (replaced by $\mathsf{sieve}, 2$) or $\mathsf{sieve}, 3, 9$ (replaced by $\mathsf{sieve}, 3$) or $\mathsf{sieve}, 2, 10$ or *etc.* The solution becomes inert when the rule sieve cannot react with any couple of integers in the solution, that is to say, when the solution contains only prime numbers. The result of the computation in our example is $\langle \mathsf{sieve}, 2, 3, 5, 7\rangle$.

To access within a sub-solution (*e.g.*, to get the result of a sub-program), a reaction rule has to wait for its inertia. That means that a reaction rule matches only inert sub-solutions. For example, if we want to compute the largest prime number lower than 10, we can use the previous program as a sub-program, *i.e.*, a sub-solution, and then compute the maximum of its result:

$$\begin{aligned}&\mathbf{let}\, \mathsf{sieve} = \mathbf{replace}\, x, y\, \mathbf{by}\, x\, \mathbf{if}\, x \text{ div } y\, \mathbf{in} \\ &\mathbf{let}\, \mathsf{max}\ = \mathbf{replace}\, x, y\, \mathbf{by}\, x\, \mathbf{if}\, x \geq y\, \mathbf{in} \\ &\langle \langle \mathsf{sieve}, 2, 3, 4, 5, 6, 7, 8, 9, 10\rangle,\ \mathbf{one}\langle \mathsf{sieve} = s,\ \omega\rangle\, \mathbf{by}\, \omega,\ \mathsf{max}\rangle\end{aligned}$$

Initially, the one-shot rule cannot react with the non-inert sub-solution, and only reactions inside the sub-solution can occur. When the sub-solution becomes inert, the one-shot rule matches the sub-solution: the variable s matches the rule named sieve and the variable ω matches all the remaining atoms of the solution (the prime numbers). In the reaction, the one-shot rule and the sub-solution are replaced by the prime numbers (ω) and the rule max which, in turn, triggers new reactions until one element remains. More formally, the execution steps occur as follows:

$$\begin{gathered}\langle \langle \mathsf{sieve}, 2, 3, 4, 5, 6, 7, 8, 9, 10\rangle,\ \mathbf{one}\langle \mathsf{sieve} = s,\ \omega\rangle\, \mathbf{by}\, \omega,\ \mathsf{max}\rangle \\ \downarrow * \\ \langle \langle \mathsf{sieve}, 2, 3, 5, 7\rangle,\ \mathbf{one}\langle \mathsf{sieve} = s,\ \omega\rangle\, \mathbf{by}\, \omega,\ \mathsf{max}\rangle \\ \downarrow \\ \langle 2, 3, 5, 7, \mathsf{max}\rangle \\ \downarrow * \\ \langle 7, \mathsf{max}\rangle\end{gathered}$$

This example shows the higher-order property of HOCL: the one-shot rule removes and adds other (named) rules. This property allows to express coordination of chemical programs within the language.

The examples provided here are simple and fine grain in order to illustrate the mechanisms of chemical programming through HOCL. But HOCL can also be used as a coordination language of services (coarse grain).

IV. Service Composition

Coordination and composition of services have attracted a lot of attention of both the industry and the academia. Several approaches have been proposed following either the orchestration or the choreography paradigms. These two paradigms differ from their execution scenario which is mainly centralized for the first one and distributed for the latter. In the following sections, we review several approaches with a particular focus on those that are based on paradigms that share some common roots with chemical programming (rewriting systems, rule-based programming, Event-Condition-Action (ECA) and tuple space).

A. Using standards

Starting from industry-led initiatives, the standard approach to compose Web Services is the Business Process Execution Language, WS-BPEL [15]. WS-BPEL is a language that provides several powerful control flow structures such as condition, loops, switches and activities, web services invocation, sequential and parallel composition. In addition, WS-BPEL provides variables to store temporary data and fault compensation. WS-BPEL is very verbose largely due to XML root and shares some similarities with assembly languages. Its level of abstraction is rather low forcing programmers to "think parallel" and to anticipate all possible failures during the workflow execution. Finally, since WS-BPEL is about orchestration, workflow execution is centralized thanks to a WS-BPEL engine. One of the main drawback of WS-BPEL is its lack of formal semantics. In [16], [17], a formal semantics of some of the WS-BPEL features, such as the specification of events, fault and compensation handler behaviors or transactions, are introduced.

Regarding choreography, the main standard today is the Web Service Choreography Description Language, WS-CDL [18]. Choreography models the interactions and dependencies between a set of services by describing their exchanges of messages. As for WS-BPEL, WS-CDL provides control structures such as sequence, parallel and choice. Loops are allowed thanks to the WorkUnit activity that provides a way to repeat the execution of an activity depending on a guard condition. As for WS-BPEL, WS-CDL is based on XML and thus is verbose with a low level of abstraction.

B. Rewriting systems

As chemical programming takes its root from rewriting systems, we can mention the work presented in [19] that describes a dynamic service customization and composition framework for Web services based on a rule-based service integration language with concepts borrowed from rewriting systems. Composition of services using an Event-Condition-Action rule based approach, that is even closer to chemical programming, is described in [20]. Workflows are expressed using a set of ECA rules. The proposed approach is easy to understood for the end-users while offering an expressive way to model the coordination of complex relationships between services.

C. Tuple space

Coordination of services using a Linda-like tuple space, similar to a multiset in chemical programming, has been investigated by several research groups. The motivation behind this idea is to allow asynchronous interactions between services providing a mean to implement the loose coupling concept associated with the service computing paradigm. In [21], the authors describe the WSSEC-SPACES coordination model that extends Linda to cope with security issues. In this approach, communications do not occur directly between services and a coordination engine but are mediated from the coordination space (tuple space). It thus allows a loose coupling in both time and space since it is not necessary to have all required services available at a given time. The results of one service can be stored into the tuple space that will trigger later the execution of a service once it will be available. The tuple space is thus used as a mechanism to transfer both control and data but in an asychronous way. The work presented in [22] goes further in this direction by using a tuple space to support self-coordination. The main rationale for self-coordination of services is to allow asynchronous interactions during the selection and the execution of services. The authors introduced the concept of control tuples to let services to adjust their execution orders and to review data and temporal dependencies at runtime.

V. Chemical Coordination and self-organisation

In [9], workflows are expressed as chemical program. It shows that all coordination mechanism of workflow can be translated into a chemical setting. The enactment of workflows can also be described by a chemical program. In fact, many classical coordination mechanisms can be expressed as a chemical coordination [23], [24]: sequential execution, parallel execution, mutual exclusion, atomicity, message passing, shared memory, rendez-vous, Kahn networks, *etc.* A very interesting property by the chemical paradigm concerns the ability of programming in a straightforward manner self-organizing systems.

A. Coordination Mechanisms

In this section, we illustrate, with two classical examples, the basic coordination facilities offered by the chemical paradigm.

1) Atomic Capture: A fundamental property of HOCL to express coordination is the atomic capture. A reaction rule takes all its arguments atomically. Either all the required arguments are present or no reaction occurs. If all the required arguments are present, none of them may take part in another reaction at the same time. Atomic capture is useful to express non blocking programs. For example, the famous dining philosophers problem can be expressed in HOCL as follows:

$$\begin{aligned} eat = {} & \textbf{replace}\, Fork{:}f1, Fork{:}f2\, \textbf{by}\, Phi{:}f1 \\ & \quad \textbf{if}\, f2 = f1 + 1 mod N \\ think = {} & \textbf{replace}\, Phi{:}f\, \textbf{by}\, Fork{:}f, \\ & \qquad\qquad Fork{:}(f+1\ mod\ N) \\ & \quad \textbf{if}\, true \end{aligned}$$

Initially the multiset contains N forks (i.e., N pairs $Fork{:}1, ..., Fork{:}N$) and the two n-shot reaction rules eat and think. The eat rule looks for two adjacent forks $Fork{:}f1$ and $Fork{:}f2$ with $f2 = f1 + 1\ mod\ N$ and produces the eating philosopher $Phi{:}f1$.

This reaction relies on the atomic capture property: the two forks are taken simultaneously (atomicity) and this prevents deadlocks. The think rule transforms an eating philosopher into two available forks. This rule models the fact that any eating philosopher can be stopped non deterministically at anytime.

2) Serialization: A key motivation of chemical models in general, and HOCL in particular, is to be able to express programs without any articial sequentiality (i.e., sequentiality that is not imposed by the logic of the algorithm). However, even within this highly unconstrained and parallel setting, sequencing of actions can be expressed. Sequencing relies on the fact that a rule needing to access a sub-solution has to wait for its inertia. The reaction rule will react after (in sequence) all the reactions inside the sub-solution have completed. The HOCL program that computes all the primes lower than a given integer N can be expressed by a sequence of actions that first computes the integers from 1 to N and then applies the rule

$$sieve\colon << iota, N >, thensieve >$$

where

$$\begin{aligned} &thensieve = \textbf{one} < iota = r, x, \omega > \textbf{by}\ sieve, \omega \\ &iota = \textbf{replace}\ x\ \textbf{by}\ x, x-1\ \textbf{if}\ x > 1 \\ &sieve = \textbf{replace}\ x, y\ \textbf{by}\ x\ \textbf{if}\ x\ div\ y \end{aligned}$$

The rule iota generates the integers from N to 1 using the notation x to denote a distinguished (e.g., tagged) integer. The one-shot rule thensieve waits for the inertia of the sub-solution. When it is inert, the generated integers are extracted and put next to the rule sieve (iota and the tagged integer 1 are removed). The wait for the inertia has serialized the iota and sieve operations. Most of the existing chemical languages share these basic features. They all have conditional reactions with atomic capture of elements.

B. Self-organization

Autonomic computing provides a vision in which systems manage themselves according to some predefined goals. The essence of autonomic computing is self-organization. Like biological systems, autonomic systems maintain and adjust their operation in the face of changing components, workloads, demands and external conditions, such as hardware or software failures, either innocent or malicious. The autonomic system might continually monitor its own use and check for component upgrades.

1) Basic principle: Chemical programming is well suited to the description of autonomic systems [8]. It captures the intuition of a collection of cooperative components that evolve freely according to some predefined constraints (reaction rules). System self-management arises as a result of interactions between members, in the same way as 'intelligence' emerges from cooperation in colonies of biological agents.

When a HOCL program reaches an inert state (*i.e.*, a stable state), and if then some new molecules are added (*i.e.*, perturbation of the system), then some new reactions happen with the new molecule until a new inert state is reached (*i.e.*, a new stable state). An autonomic system as depicted in Figure 2 can thus be represented using HOCl by external input/output operations into the multiset for the *Collect* and *Act* steps whereas the *Analyze* and *Decide* steps are represented by applying the rules on data stored in the multiset. External events can trigger input operation by storing new elements into the multiset while the presence of a certain type of elements can generate external events. A simple mail system [8] has been developed as an example of programming self-organization with HOCL. It features self-healing, self-optimizing, self-protection and self-configuration.

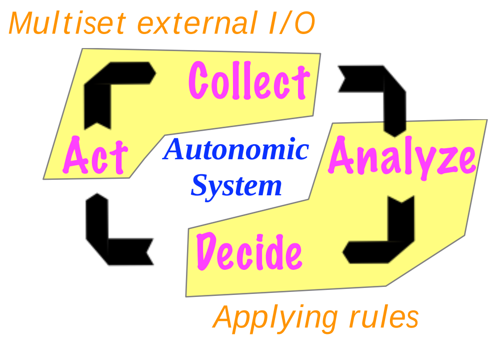

Figure 2. Autonomicity with HOCL

2) A Self-Organizing Sorting Algorithm: Consider the general problem of a system whose state must satisfy a number of properties but which is submitted to external and uncontrolled changes. This system must constantly reorganize itself to satisfy the properties. Let us illustrate this class of problem by a simple sorting example where the system state is made of pairs (index : value) and the property of interest is that values are well ordered (*i.e.*, a smaller index means a smaller value). If the environment keeps adding random pairs to the state, the system must reorganize itself after each insertion of an ill-ordered element. The system is represented by a chemical solution $State =< sort, (i1{:}v1), ..., (in{:}vn) >$, consisting of pairs and the following active molecule:

$$\begin{aligned} sort = \textbf{replace}(i{:}x), (j{:}y)\ \textbf{by}(i{:}y), (j{:}x) \\ \textbf{if}\ i < j\ and\ x > y \end{aligned}$$

The molecule sort looks for couples of ill-ordered values and swaps them. The solution evolves up to the point where no more reactions are possible: the solution has reached a stable state and the ordering property is satisfied.

Adding new ill-ordered values breaks the 'equilibrium', since the ordering property is violated. However, sort searches continuously for new ill-ordered values and causes reactions so that the state will reach a new stable state. This very simple example shows how chemical programming can naturally express self-organizing systems. A program is made of a collection of rules (active molecules), which react until a stable state is reached and the corresponding invariant properties satisfied. These rules remain present and are applied (without any external intervention) as soon as the solution becomes unstable again.

C. Applications

HOCL as a coordination language has also been applied to program Desktop Grids. A Desktop Grid is made of non dedicated resources (*e.g.*, any personal computer connected on the Internet). Such a grid can be highly volatile and non reliable. In [11], HOCL is used as a coordination language to specify the execution of a simple ray-tracer in a Desktop Grid. The HOCL program contains rules that adapt the on-going executions of programs according to the availability of resources in the Desktop Grid.

An autonomic mail system has been described within the chemical framework [8]. Rules ensure that all messages have reached their destination; if it is not the case some reactions occur to reach that state. The system includes rules that ensure other autonomic properties such as self-healing (rules that set and unset emergency servers in case of failures), self-optimization (rules that balance the load between several servers), self-protection (rules that suppress spam or viruses), self-configuration (rules that forward messages to the new address of a user) and so forth.

We have also specified a chemical Distributed Versioning System [7]. In this application, several editors may concurrently edit a document consisting of a set of files. The editors are distributed over a network and each works on a different version, making modifications to the files and committing them locally. From time to time, two or more editors merge their modifications to propagate them. This system can be easily described with reaction rules to reflect the self-organization of the system. The system is also self-repairing: if an editor loses his local version it can revert to a previous state by synchronizing with one or several other editors.

VI. Service Orchestration Using HOCL

A service orchestration is a program that describes a coordination of services. HOCL can be used as a data-driven coordination language according a chemical metaphor. For example, the previous HOCL examples can be viewed as data-driven coordination of integers and of functions on integers (div, $\geq$). A chemical service architecture consists of a solution of services, *i.e.*, a solution of sub-solutions, each sub-solution representing one service:

Service oriented computing	**Chemistry**
Services	Sub-solutions
Service calls	Molecules
Orchestration	Chemical reactions

A service is represented by a sub-solution that contains molecules performing the operations that this service proposes. Figure 3 shows the two basic rules to perform a service call. A molecule of the form Call:s:n:p in the solution represents the called service, where s is the calling service, n is the identifier of the call for the calling service, and p are the parameters for the operation to be performed.

```
let withdrawServiceCall =
      replace serv1:⟨ExtCall:serv2:n:param, w⟩
          by serv1:⟨w⟩, Call:serv1:serv2:n:param
in
let depositServiceCall =
      replace Call:serv1:serv2:n:param, serv2:⟨w⟩
          by serv2:⟨Call:serv1:n:param, w⟩
in
⟨withdrawServiceCall, depositServiceCall,
 Service1:⟨...⟩, ..., ServiceN:⟨...⟩
⟩
```

Figure 3. Calling a service.

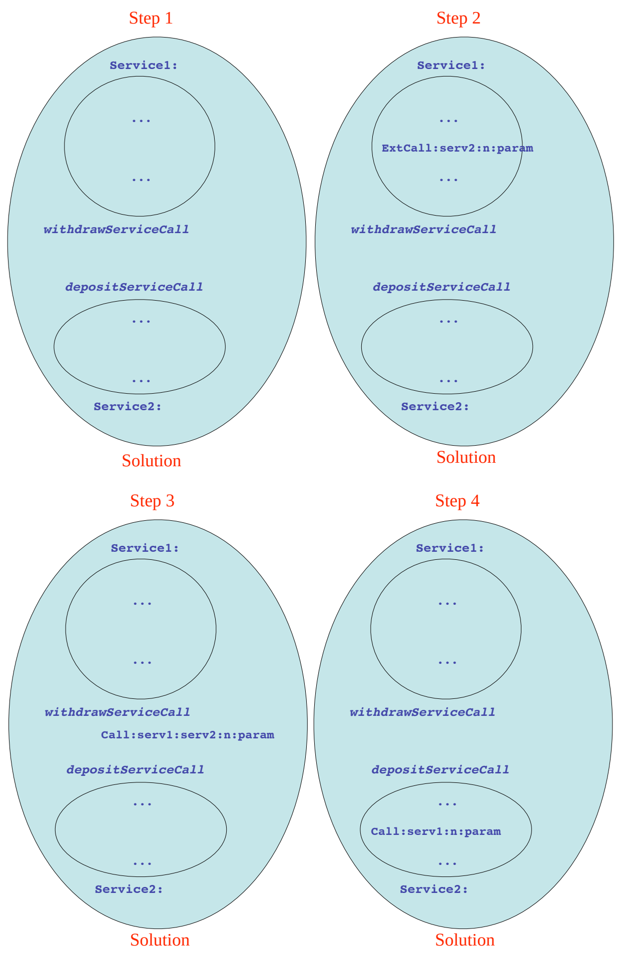

Figure 4. Calling a service with HOCL

When a service makes a call to another service it generates a molecule of the form ExtCall:s:n:p where s is the called service, n is the identifier of the call for the current (calling) service, and p are the parameters for the operation to be performed.

At a given time, a service may be running different computations related to different simultaneous call. That's why, to prevent a service to mix the computation and the result of different and independent calls, each call has a unique identifier n.

A call is performed in several steps by two rules (see Figure 4). The rule withdrawServiceCall extracts an ExtCall molecule from a service sub-solution and puts it in the main solution. The rule depositServiceCall takes a call in the main solution and forwards it into one corresponding service sub-solution.

Two remarks:

- A call to a service to perform an action, and a call to a service to provide a result are just two ordinary calls: there is no distinction between the two calls. In fact, a call message and its result message are not explicitly coupled like in a RPC for example.
- According to the semantics of HOCL, a rule may react only with an *inert* solution. So the rules withdrawServiceCall and depositServiceCall could only react with inert sub-solutions, *i.e.*, with services that do not do any computation, *i.e.*, with services that only perform communication (input or output of calls). In fact, that inertia constraint may be released for these two rules. These two rules manage molecules that represent messages. So these molecules are independent of computations happening inside the solutions representing the services. Adding or removing a call from these solutions does not depend on the internal state of these solutions. So adding a call to or removing a call from a solution that represents a service can happen even if the solution is not inert.

1) A travel organizer example: Let's take the example of a travel organizer (cf Figure 5) which makes the reservations of a flight and a hotel according to some parameters provided by a user.

```
let findFlightHotel =
        replace Call:s:m:p, n
             by ExtCall:FlightService:n:param,
                ExtCall:HotelService:n:param,
                s:m:n, (n+1)
in
let resultFlightHotel =
        replace Call:FlightService:n:f,
                Call:HotelService:n:h,
                s:m:n
             by ExtCall:s:m:(f:h)
in
TravelOrgService:⟨0, findFlightHotel,
                      resultFlightHotel⟩
```

Figure 5. A travel organizer service in HOCL.

The service is a solution named TravelOrgService. It contains an integer used as a counter to provide unique identifiers to distinguish different molecules related to different calls. For each call to the travel organizer service, the rule findFlightHotel generates two calls: one call to a flight service and one call to a hotel service. It also updates the counter, and stores the reference to that call as a molecule $s{:}m{:}n$ where $s{:}m$ identifies the calling service, and n is the identifier to this call to the travel organizer service. The rule resultFlightHotel reacts when the results are available. The results appear as two calls: one from a flight service, and one from a hotel service. They both concern the same initial call identified by the counter n. Then the rule generates a call back to the calling service s with its identifier m, and the flight and hotel results $f{:}h$.

2) Example of an execution: We describe here a possible execution of the travel organizer example (cf Figure 6). The system is represented by a solution that contains the rules withdrawServiceCall and depositServiceCall that perform the calls, the sub-solutions representing the services (the travel organizer service, the flight services and the hotel services), and two calls to the travel organizer by two different users.

```
⟨withdrawServiceCall, depositServiceCall,
 TravelOrgService:⟨findFlightHotel⟩,
 FlightService:⟨Name:AirFrance,...⟩,
 FlightService:⟨Name:BritishAirways,...⟩,
 HotelService:⟨Name:Accor,...⟩,
 ...,
 Call:UIService1:TravelOrgService:n1:
                 (Dates1:Places1:Pref1),
 Call:UIService2:TravelOrgService:n2:
                 (Dates2:Places2:Pref2)
⟩
```

Figure 6. Running the travel organizer service in HOCL.

Two users have queried a search for their travel and the system has added the respective calls of the form Call:UIServiceX:TravelOrgService:... into the main solution, where UIServiceX is the identifier of the user interface service that has emitted the call to the travel organizer service TravelOrgService, where DatesX, PlacesX are the dates and places constraints for the required travels, and PrefX some additional preferences. Initially, the rule depositServiceCall can forward the two calls to the travel organizer service. Then the travel organizer will generate the calls to the flight services and the hotel services using the rule findFlightHotel. Then the rule withdrawServiceCall will extract the calls to the main solution, and the rule depositServiceCall will forward these calls to a corresponding service. After, some reactions these services will generate their result inside their solution as an ExtCall molecule addressed to the travel organizer. The rules withdrawServiceCall and depositServiceCall will then bring these messages to the travel organizer. When both results from the flight service and the hotel service are available inside the TravelOrgService for the same initial call, the rule resultFlightHotel generates an external call to the service that invoked the travel organizer service. Eventually, the rule withdrawServiceCall will extract that result from the travel organizer service solution and put it in the main solution, so that it is available to the external world (outside the main solution). At any time, at run time in particular, some new services may be added inside the main solution, and some services may be removed from the main solution. This is not a problem, since the coupling between a call and a corresponding service is dynamic and non deterministic. The service type of the called service must be satisfied: for example, several services provide a flight service, and a call to a service flight may react with any of them.

3) Calling multiple instances of a service: In the previous example (Figure 6), TravelOrgService cannot interact with several instances of a specific service such as FlightService and HotelService. There is no way to chose a particular instance depending on QoS constraints. Either the solution does not contain any service when starting the chemical program and then when the first service will appear in the solution it will be selected or there are already several instances of a particular service (which is the case in Figure 6 with FlightService), and the decision will be taken in a non deterministic way and only a particular instance of a service will be called. As a consequence, it is not possible to compare several travel plans and to select the one that fits well user's wishes based on a set of constraints (costs, travel time, ...). To do so, it is necessary to call several services to know the availability of flights and hotel rooms. To allow interacting with several instances of a service, which means calling several services instead of a single one, it is only required to change one of the two rules that perform the service call (Figure 3) illustrating the flexibility provided by the chemical model. Only the depositServiceCall rule needs a slight modification to be able to perform multiple calls instead of a single one as shown in Figure 7.

```
let withdrawServiceCall =
        replace serv1 : ⟨ ExtCall : serv2 : n : param , w⟩
             by serv1 : ⟨w⟩ ,
                Call : serv1 : serv2 : n : param
in
let depositServiceCall =
        replace Call : serv1 : serv2 : n : param ,
                serv2 : ⟨ State : AVAIL , w⟩
             by Call : serv1 : serv2 : n : param ,
                serv2 : ⟨ Call : serv1 : n : param ,
                State : CALLED , w⟩
in
⟨ withdrawServiceCall , depositServiceCall ,
  Service1 : ⟨ State : AVAIL , . . . ⟩ ,
  . . . ,
  ServiceN : ⟨ State : AVAIL , . . . ⟩
⟩
```

Figure 7. Calling multiple instances of a service.

The basic idea to generate multiple calls is to leave the molecule Call:*serv1*:*serv2*:*n*:*param* in the main solution instead of removing it when applying the depositServiceCall and to add a molecule, tagged State, within the sub-solution representing the service. The State molecule can be either AVAIL or CALLED. Once the service is called the State molecule is set as CALLED meaning that the service has already been called and will thus avoid the rule depositServiceCall to be shot again using the same service instance. The proposed approach can deal with services already in the solution, those that have been already discovered before the execution of the chemical program, and new services discovered at runtime. After issuing a call to multiple instances of a service, the results have to be sent back to the caller. A set of rules, stored in the sub-solution representing the caller, have to be specified to select those that are relevant depending on a set of constraints.

VII. Conclusion and Future Work

Originally, the Gamma formalism was invented as a basic paradigm for parallel programming [14]. It was proposed to capture the intuition of a computation as the global evolution of a collection of atomic values evolving freely. Gamma appears as a very high level language which allows programmers to describe programs in a very abstract way, with minimal constraints and no articial sequentiality. Later, it became clear that a necessary extension to this simple formalism was to allow elements of a multiset to be Gamma programs themselves, thus introducing higher-order. This lead to the HOCL language used in this paper.

Basically, the chemical paradigm (as introduced in HOCL) offers four basic properties: mutual exclusion, atomic capture, parallelization and serialization. These properties have been exploited in [23] in order to give a chemical expression of well known coordination schemes.

Along the same lines, this paper investigates the utilization of the Chemical Programming Model, in order to describe Service Coordination. We develop this idea with a simple, yet practical, example dealing with travel organization. The example developed throughout section VI shows that our approach provides a very abstract and generic way of programming service orchestration. This is made possible due to the higher order property of HOCL. Programs (services) can be handled naturally by appropriate synchronization rules. Here, we have limited our investigations to service orchestration, it is clear that we could have tackled more elaborated synchronization schemes dealing with service choreography.

Another issue, that is not covered by the paper, is the computational complexity of chemical programs and thus, in the context of this paper, how our approach can scale with respect to the number of managed services. This is of course an important issue but this problem has already been addressed in a more general context and several solutions have been proposed to keep the computational complexity acceptable. One of these solutions is based on a structured multiset using types [25] that provides a way to specify relations between molecules. These relations can be used to consider only a subset of the molecules when shooting a rule instead of all molecules within a solution. Another approach, based on spatial notions, has also been investigated in [26]. Eventually, by performing a static analysis of a chemical program, it is often possible to limit drastically the number of useless checks for reaction and thus, the number of reactions. Of course, those solutions have to be adapted to the context of service coordination using the chemical metaphor, in particular the definition of neighboring relations between services. As a future work, we plan to further investigate this problem in order to offer a scalable coordination approach using the chemical metaphor.

Acknowledgement

We would like to thank Pascal Fradet and Yann Radenac for their invaluable help with this paper.

References

[1] A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls," *ACM Trans. Comput. Syst.*, vol. 2, no. 1, pp. 39–59, 1984.

[2] OMG, "The Common Object Request Broker: Architecture and Specifica tion V3.0," Object Management Group, Tech. Rep. OMG Document formal/02-06-33, June 2002.

[3] Open Management Group (OMG), "CORBA components, version 3," Object Management Group, Document formal/02-06-65, June 2002.

[4] I. Foster and C. Kesselman, Eds., *The Grid 2: Blueprint for a New Computing Infrastructure*, 2nd ed. Morgan Kaufmann Publishers, 2003.

[5] "Google app engine. http://code.google.com/appengine."

[6] "Amazon services. http://aws.amazon.com."

[7] J.-P. Banâtre, P. Fradet, and Y. Radenac, "Generalised multisets for chemical programming," *Mathematical Structures in Computer Science*, vol. 16, no. 4, pp. 557–580, Aug. 2006.

[8] ——, "Chemical specification of autonomic systems," in *Proc. of the 13th Int. Conf. on Intelligent and Adaptive Systems and Software Engineering (IASSE'04)*, 2004.

[9] Z. Németh, C. Pérez, and T. Priol, "Workflow enactment based on a chemical metaphor," in *The 3rd IEEE International Conference on Software Engineering and Formal Methods*, September 2005.

[10] ——, "Distributed workflow coordination: Molecules and reactions," in *The 9th International Workshop on Nature Inspired Distributed Computing*. IEEE, 2006, p. 241.

[11] J.-P. Banâtre, N. Le Scouarnec, T. Priol, and Y. Radenac, "Towards "chemical" desktop grids," in *Proceedings of the 3rd IEEE International Conference on e-Science and Grid Computing (e-Science 2007)*. IEEE Computer Society Press, 2007.

[12] "Web service resource framework. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrf."

[13] J.-P. Banâtre, P. Fradet, and Y. Radenac, "Principles of chemical programming," in *Proceedings of the 5th International Workshop on Rule-Based Programming (RULE 2004)*, ser. ENTCS, S. Abdennadher and C. Ringeissen, Eds., vol. 124, no. 1. Elsevier, 2005, pp. 133–147.

[14] J.-P. Banâtre and D. Le Métayer, "Programming by multiset transformation," *Communications of the ACM (CACM)*, vol. 36, no. 1, pp. 98–111, Jan. 1993.

[15] C. Barreto, V. Bullard, T. Erl, J. Evdemon, D. Jordan, K. Kand, D. Knig, S. Moser, R. Stout, R. Ten-Hove, I. Trickovic, D. van der Rijn, and A. Yiu, "Web services business process execution language version 2.0," http://www.oasis-open.org/committees/wsbpel, May 2007.

[16] M. Mazzara and S. Govoni, "A case study of web services orchestration," in *Proc. of the 7th International Conference on Coordination Models and Languages, Lecture Note in Computer Sciences*, vol. 3454, Dec 2005. [Online]. Available: http://www.springerlink.com/index/yvq99geyaktgu39g.pdf

[17] R. Lucchi and M. Mazzara, "A pi-calculus based semantics for ws-bpel," *Journal of Logic and Algebraic Programming*, January 2007.

[18] S. Ross-Talbot and T. Fletcher, "Web services choreography description language: Primer," Jun 2006. [Online]. Available: http://www.w3.org/TR/2006/WD-ws-cdl-10-primer-20060619/

[19] J. Chen, "Rewrite rules as service integrators," in *Proceeedings of the Rules And Rule Markup Languages For The Semantic web workshop (RuleML 2004), Lecture Note in Computer Sciences*, Dec 2004, pp. 182–187.

[20] L. Chen, M. Li, and J. Cao, "A rule-based workfow approach for service composition," in *Third International Symposium Parallel and Distributed Processing and Applications (ISPA)*, ser. Lecture Notes in Computer Science, Springer, Ed., vol. 3758, October 2005, pp. 1036–1046.

[21] R. Lucchi and G. Zavattaro, "Wssecspaces: a secure data-driven coordination service for web services applications," *Proceedings of the 2004 ACM symposium on Applied computing*, Jan 2004. [Online]. Available: http://portal.acm.org/citation.cfm?id=968001

[22] Z. Maamar, D. Benslimane, C. Ghedira, Q. H. Mahmoud, and H. Yahyaoui, "Tuple spaces for self-coordination of web services," in *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*. New York, NY, USA: ACM, 2005, pp. 1656–1660.

[23] J.-P. Banâtre, P. Fradet, and Y. Radenac, "Classical coordination mechanisms in the chemical model," in *From semantics to computer science: essays in honor of Gilles Kahn*. Cambridge University Press, 2009.

[24] ——, "The chemical reaction model, recent developments and prospects," in *Software-Intensive Systems and New Computing Paradigms*, ser. Lecture Notes in Computer Sciences (LNCS), State-of-the-Art, Survey, M. Wirsing, J.-P. Banâtre, M. Hölzl, and A. Rauschmayer, Eds., no. 5380. Springer, September 2008, pp. 209–234.

[25] P. Fradet and D. Le Métayer, "Structured gamma," *Sci. Comput. Program.*, vol. 31, no. 2-3, pp. 263–289, 1998.

[26] J.-L. Giavitto, O. Michel, J. Cohen, and A. Spicher, "Computation in space and space in computation," in *Unconventional Programming Paradigms (UPP'04)*, ser. Lecture Notes in Computer Sciences (LNCS), State-of-the-Art, Survey, J.-P. Banâtre, P. Fradet, J.-L. Giavitto, and O. Michel, Eds., no. 3566. Springer, September 2005, pp. 137–152.