Using Chemical Metaphor to Express Workflow and Service Orchestration

Chen Wang, Jean-Louis Pazat

2010 10th IEEE International Conference on Computer and Information Technology (CIT 2010)

Using Chemical Metaphor to Express Workflow and Service Orchestration

Chen WANG and Jean-Louis PAZAT
INRIA/IRISA, Campus de Beaulieu, 35042, Rennes Cedex, France
{chen.wang, jean-louis.pazat}@irisa.fr

***Abstract*—Nowadays, novel applications, such as personalized e-commerce services, call for cooperation across enterprise boundaries. Service-Oriented-Architecture (SOA) forms a solution to build loosely coupled distributed applications by composing Web services that are provided by different entities. Orchestration is a perspective of Web service composition. The traditional approach to implement orchestration is to use an executable language such as WS-BPEL to express a workflow. However, it has some undesirable drawbacks due to its static nature. This paper explores an unconventional approach known as chemical programming on expressing workflow and service orchestration. It has some valuable characteristics such as automated, dynamic and parallel execution which enable it to be a competitive candidate for service composition.**

***Index Terms*—Web service; service orchestration; workflow; chemical programming; HOCL; SOA;**

I. Introduction

Service-Oriented Architecture (SOA), as an architectural platform, is adopted today by many businesses as an effective approach for building software applications that promotes loose coupling between software components [1]. Although the term "service" is widely used today, it is hard to find a unique definition. Within the scope of this paper, a service is a software system designed to support interoperable machine-to-machine interaction over computer networks. It can be defined within the scope of an enterprise, implemented in different programming languages, run on various platforms and operating systems.

Orchestration [2] is a perspective of Web service composition. It aims at providing an open, standards-based approach for connecting Web services together to create higher-level business processes [3]. An orchestration describes the interaction with both internal and external Web services at the message level. There is a centralized control for message exchanges and execution logic. This control is always from one party's viewpoint.

The most common way to build composite applications is to use an executable language (such as WS-BPEL [4]) to create a business process by defining a workflow that specifies business logic and execution order. A centralized orchestration engine is implemented to execute this business process. It invokes and coordinates all the partner services by message passing in order to achieve its promising goal.

However, WS-BPEL, the most widely used service orchestration language, has some undesirable drawbacks. Firstly, it is naturally static. If a process definition is accepted by the orchestration engine, it can not be modified during run-time. Furthermore, it lacks of formal semantics, which makes it difficult to formally reason a process behavior [5]. Thirdly, a centralized execution engine is implemented which may cause the challenge of scalability, reliability and availability. Finally, its XML-based grammar is complicated and error-prone; it also has some limitations in expressing sophisticated and complex business workflow. To deal with all these problems, this paper explores an innovative approach known as chemical computing paradigm on expressing Web service orchestration.

Chemical programming [6], [7] is an unconventional programming paradigm which is inspired by the chemical metaphor. Computation can be seen as chemical reactions where all the molecules involved represent computing resources. The reactions are controlled by a set of rules which are similar to chemical equations: a chemical equation reflects a nature law which specifies the reactant and resultant of a chemical reaction; and a rule defines the input and output of a certain computation.

Each chemical program creates a multi-set [8] acting as a chemical reaction container. All the elements such as computing resources and rules are defined inside as molecules. Once a rule gets all its required inputs and a certain condition prevails, it will then be activated - those input elements will be replaced by new ones according to the rule definition. Using chemical reaction as a metaphor, once two react-able molecules meet together and a certain temperature is reached, according to a certain chemical equation, new substance will be produced.

Compare to the traditional computing paradigms, chemical computing has the following superiorities: first of all, it is highly dynamic, when the multi-set reaches a stable state that is called "inert", new rules and resources can be added into the multi-set to trigger new reactions; furthermore, it is automated and self-coordinated, no intervene is needed during the computing process. All these characteristics make us believe that chemical computing model is suitable for service orchestration programming.

This paper is organized as follows: section II gives a brief introduction to chemical programming models. In section III we propose a framework based on chemical programming paradigm to express workflow and service orchestration. Section IV presents a prototype which implements the proposed framework. In Section V, some of the main existing approaches to express service orchestration are introduced. Finally, we draw a conclusion and address the future work

978-0-7695-4108-2/10 $26.00 © 2010 IEEE
DOI 10.1109/CIT.2010.268

in Section VI.

II. Chemical Programming

A. Gamma

Gamma is proposed in [9] as a paradigm for parallel computing through a chemical metaphor. Computation in Gamma can be seen as a series of chemical reactions; data involved in the computation is represented by molecules. The multi-set is the unique data structure that can be regarded as the chemical reaction container. Multi-set extends the concept of set with multiplicity. An element can be presented only once in a set whereas many times in a multi-set. The time of its occurrence is defined as multiplicity. For example, {1, 3, 1} is a multi-set while not a set because the multiplicity of element "1" is 2.

Computation in Gamma is performed by multi-set rewriting [8] which is controlled by a set of rules. A rule is composed of 2 parts: *condition* and *action*. If a portion of elements satisfy the *condition*, they will be replaced by other elements specified by *action* part. These newly produced elements may activate another rule and trigger new reactions; this "domino process" will continue until the container become inert - that is to say, no element in the multi-set can trigger any reactions. When such a stable state is reached, the final result is obtained.

As a short example, to compute the maximum number in a non-empty multi-set, the following rule is defined:

replace x, y by x if x>= y

And then add this rule into the following multi-set: {1, 2, 4, 6, 5, 7, 8, 9}. x, y can match any pair of integers, i.e., x=4, y=1, hence 1 will be removed from the multi-set. In this way, only the maximum number will be finally left in the multi-set.

B. HOCL

HOCL stands for Higher-Order Chemical Language. It implements γ-calculus [10] which can be seen as an higher-order extension of Gamma language. An HOCL program is composed of two parts: *rule definition* and *multi-set organization*. A sample HOCL program is defined in Figure 1 to select eligible scholar for a scholarship. The selected student should have the best average score among all the candidates whose average scores are over 18.00. All the rules are defined from *line 1 - line 10* and the elements are organized in a multi-set from *line 11 - line 20*.

A rule can be either *one-shot* or *N-shot*, the difference lies in that after the reaction, the *one-shot* rule will be consumed but *N-shot* rules can be reused repetitively. Using HOCL, *N-shot* rules are defined by "let...replace...by...if..." expression. This expression has 4 keywords. "let" specifies a name to a rule. "replace...by..." defines the multi-set rewriting to perform the calculation. "if" denotes the condition of the reaction. *One-shot* rule is defined in the liked manner by using "replace-one" key word and it is not needed to specify a name to a *one-shot* rule. In this program, two *N-shot* rules are defined: *selectCandidate* selects the qualified candidates whose average scores are above 18.00 and *searchScholar* searches the scholar

```
1  let selectCandidate =
2    replace stu::String:score::double, ?w
3    by w
4    if score < 18.00
5  in
6  let searchScholar =
7    replace stu1::String:score1::double,
         stu2::String:score2::double
8    by stu1:score1
9    if score1 >= score2
10 in
11 <
12   <
13     selectCandidate, searchScholar,
14     "Thierry":17.96,
15     "Nicola":16.98,
16     "Ameli":17.26,
17     "Christina":12.89
18   >,
19   replace-one <selectCandidate =x,
           searchScholar =y, ?w> by w
20 >
```

Fig. 1. Sample HOCL code: search eligible scholar

with the highest score; on the other hand, a *one-shot* rule is defined at *line 19* to extract the final result.

HOCL extends the previous model with pairs, types, empty solution and naming, we introduce shortly these extensions here, refer to [11] for a more elaborate introduction.

Naming. In HOCL, a rule can be named (or tagged) to facilitate its reuse. The "let" key word is used to assign a name to a rule in the following way:

let *RuleName* = ... (rule definition)

Once defined with a name, a rule can be easily reused by calling its name through this program.

Variable types. In HOCL, a variable has to be defined with a certain type. It can be either a primitive type such as *int* or *String*, or some user defined types [11]. Every variable is defined within "replace" expression in the following way:

VariableName::*Type*

Two colons are used together ("::") to indicate the type of a variable. Take the program in Figure 1 for example, in the definition of rule *selectCandidate*, a *pair* has been defined at *line 2* (stu::*String*:score::*double*). This *pair* comprises a String variable *stu* and a float variable *score*.

After the definition of variables in "replace" statement, they can be used in the following parts such as "by" and "if" statements. The scope of a variable lasts within the definition of a rule. A rule can never use variables defined by other rules.

Pairs. A pair is denoted by "A_1:A_2", a colon is used to connect two isolated molecules. This is easy to think using the

chemical metaphor. A S_2O_4 compound molecule can be seen as a simple combination of two SO_2 molecules (SO_2:SO_2). As a result, a pair can be regarded as a compound molecule which can also participate in the chemical reactions. In the program defined in Figure 1, a pair is used to store a student's name and his score (i.e..: stu_1:$score_1$ at *Line 8*).

Generally, the notion of *pair* can be extended by using several colons to connect multiple fields. As a result, a pair can be defined in the following form:

$$field_1{:}field_2{:}...{:}field_n$$

Each field is a variable and has to be defined with a certain type. In this case, a pair is also called a *tuple*. A *tuple* can be regarded as a *struct* in C language, which, like an union, groups multiple variables into a single record.

Universal type. In HOCL, there is a universal pattern defined by a question mark as "?var_name". It can match any type of variable, such as an *integer*, a *pair* or even an "empty molecule". Furthermore, it can match not only one molecule but many of them. This pattern means: "The remaining molecule(s)".

In this program, we have such a variable defined at *line 2* ("?w"). When the rule *selectCandidate* is applied for the first time, there are four *pairs* in the solution, *stu:score* can match any one of them and *w* matches the other three. This process is shown in *Figure 2*. In this case, *w* means "all the three remaining pairs".

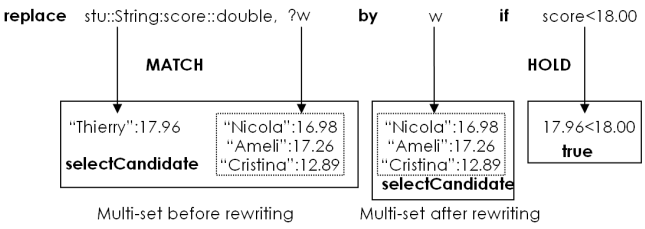

Fig. 2. Universal pattern: match multiple molecules

This process will continue until there is only one *pair* left in the multi-set, we assume "Ameli":17.26 is that pair (because the execution of a chemical program is highly non-deterministic, different executions may have different execution orders. Therefore, another *pair* can be left to the end for another execution). Then rule *selectCandidate* is applied again. *stu:score* will match "Ameli":17.26; at this moment, there is no more element left in the multi-set, *w* can only match an "empty molecule", as shown in *Figure 3*. In this case, *w* refers to an empty molecule which means "nothing is remained".

The universal pattern is meaningful since the multi-set contains a lot of molecules, but rewriting is usually performed within a certain part of them. Since not all elements need to participate in the multi-set rewriting, a universal pattern is used to package all those irrelevant elements that will keep unchanged during the reaction.

Empty solutions. The notion of empty solution in HOCL

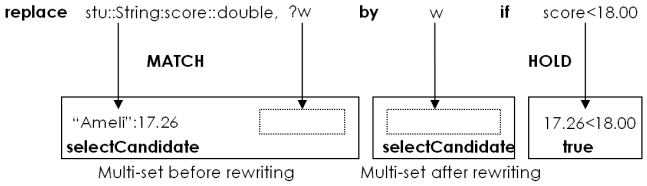

Fig. 3. Universal pattern: match "empty molecule"

is raised since that after reactions, the multi-set might become empty. This is caused by the universal pattern which can match any molecules even the empty ones. In the program shown in Figure 1, rule *selectCandidate* removes a student *pair* if his average score is lower than 18.00; in this case, no student can meet this requirement. As a result, all the pairs will be removed in the end. And then a *one-shot* rule will remove both *N-shot* rules (*selectCandidate* and *searchScholar*). Therefore, an empty solution is finally obtained. The execution sequence is shown in Figure 4.

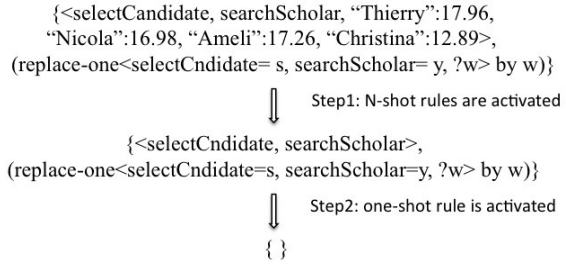

Fig. 4. Execution order

From this example, we can summarize that as a chemical programming language, HOCL has the following properties: firstly, its execution is non-deterministic; different executions can have different execution sequences, but finally the same result is obtained. Furthermore, its execution is parallel; various reactions can be carried out simultaneously. Thirdly, the execution controlled by a set of rules is automated and self-coordinated. These characteristics make us believe that HOCL is suitable for service orchestration programming, which is also full of uncertainty.

III. Service Orchestration

In this section, a framework is proposed based on chemical programming to express workflow and service orchestration.

A. Framework

In chemical programming, each compute unit is represented by a multi-set. A multi-set is also vividly called a "solution"; it acts as a membrane which blocks the molecules inside it to react with the ones outside it. As a result, it is reasonable to use a solution to represent a component which encapsulates certain functionalities, such as a service or an activity. Based on this point of view, a chemical framework is proposed to

express service orchestration, as shown in *Figure 5*. In this framework, all the elements such as computing resources and rules are organized into three hierarchical solution levels.

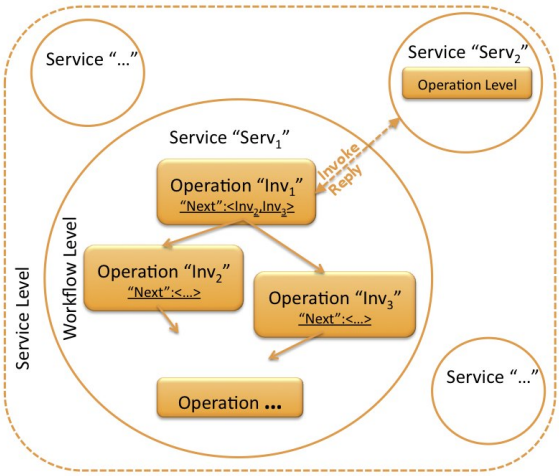

Fig. 5. Chemical Framework for Orchestration

The first level solution is the out-most multi-set of an HOCL program. It is named as *container solution* since it provides a public place for all providers to advertise their services (in Figure 5, the *container solution* is represented by the largest rectangle with rounded corners). It composes multiple sub-solutions representing services as well as some rules to perform the message passing between services; therefore, this level is also called *service level*.

The second level is *workflow level*. Workflow is defined by chemical constructs within a *service solution*. A *service solution* is a sub-solution in the container which represents a service. In Figure 5, each circle stands for a *service solution*. WS-BPEL defines workflow by coordinating activities. In chemical framework, activities are represented by the sub-solutions of a *service solution*. The relationship among the activities such as execution order are realized by a set of rules that are defined to coordinate partner services.

Since an activity usually defines an operation, these sub-solutions are also called *operation solutions*. In Figure 5, each solid rectangle stands for an activity, it contains molecules for carrying out a certain operation. This is the third level of the framework - *operation level*.

The HOCL source code in Figure 6 implements the framework in Figure 5. In the *container solution*, each service is implemented by a tuple with two fields: the first field is a string indicating the name of the service and the second field is a multi-set implementing *service solution*. The detailed implementation of "$Serv_1$" *service solution* is given out, it composes some "invoke" activities such as "Inv_1". These activities are also represented by a tuple with two fields: a string as the name and a multi-set as the *operation solution*.

In the following sections, based on this implementation of chemical framework, a set of rules are defined in both

```
< //Container solution
  "Serv1":< //Service solution
     "Inv1":< //Operation solution>,
     "Inv2":<...>, "Inv3":<...>, ...
  >,
  "Serv2":<...>,...
>
```

Fig. 6. Implementation of chemical framework in HOCL

workflow level and *service level* to express workflow and service orchestration.

B. Workflow Level Coordination

As introduced, a composite service can be built by composing a workflow . When receiving an invocation, the execution will start from the first step(s) of its workflow. A workflow composes a set of activities to coordinate partner services. In this section, we introduce how to use HOCL to express workflow within a *service solution*. We demonstrate that most of WS-BPEL constructs can be expressed by using chemical computing paradigm.

Invoke. Once a service wants invoke a partner service, it has to compose an "invoke" activity in its workflow. In chemical computing framework, to invoke a partner service, a *service solution* has to generate an *operation solution* containing a tuple molecule in the form of `"CALL":t`, where "CALL" is an identification suggesting that this is an invocation tuple and t stands for the name of the wanted partner service. In the framework shown in *Figure 5*, if the activity "Inv_1" of "$Serv_1$" wants to invoke "$Serv_2$", it has to produce the following tuple in "Inv_1" *operation solution*, `"CALL":"Serv2"`.

This "CALL" tuple will then react with rule *invoke* defined in Figure 7. When the execution reaches a certain activity, all the input parameters are packaged into an `"INPUT"` tuple and passed into the relative operation solution. This `"INPUT"` tuple will react with that `"CALL"` tuple according to the rule *invoke*. A new `"CALL"` tuple with four fields is generated in *service solution*, it implies that the activity *s* generates this invocation to service *t* with parameters *p*. The presence of this `"CALL"` tuple in the *service solution* will activate a pair of service level rules which implement the message passing between services (rf. Section III-C).

```
let invoke =
    replace s:<"CALL":t, "INPUT":<?p>,?w>
    by s:<w>, "CALL":s:t:<p>
in...
```

Fig. 7. Chemical rules for "invoke" activity

Receive. On "workflow level", there are a pair of rules defined as shown in *Figure 8* to perform "receive" activity. Once a message is arrived, it can be either an invocation or a reply. If it is an invocation (receive a "CALL" tuple), execution has to be started from the first step(s). Those starting steps

are indicated by a "START" tuple in the *service solution*. Rule *receiveCall* will package parameters into an "INPUT" tuple and then move it to the operation solution(s) of those starting step(s). In this way, the execution of workflow starts. As for the remaining parts of this "CALL" tuple such as *s* and *f* fields, the partner service will create a "CLIENT" tuple suggesting the invoker who has generated this invocation. After the calculation, these information will be used to locate the invoker to return the result.

```
let receiveCall =
    replace "CALL":s:f:<?P>, "START":<t>,
         t:<?w>
    by t:<w, "INPUT":<P>>,"CLIENT":s:f
in
let receiveReply =
    replace "REPLY":s:<?R>, s:<?w>
    by s:<w,"RESULT":<R>>
in...
```

Fig. 8. Chemical rules for "Receive" activity

On the other hand, a message can also be a reply (receive a "REPLY" tuple, rf. next point). In this case, the parameter has to be sent to the exact step which has generated the respective invocation. Rule "*receiveReply*" defined in Figure 8 will repackage the result in a "RESULT" tuple and send it to the respective sub-solution(s). Each sub-solution defines rules to process the result, either to pass it to the next execution step(s), or package it into a "REPLY" tuple to send back to the client.

Reply. Once a partner service finishes its calculation for a client, a "reply" activity is created to return the results. The "reply" activity is implemented in the same way as "invoke"; Once a service finishes its calculation, it encapsulates the result in a "RESULT" tuple in its *service solution*. Remember when receiving a message invocation, the invoker's information is saved in a "CLIENT" tuple. Rule *reply* catches these two tuple and generate a "REPLY" tuple in the form of `"REPLY":s:f:<R>`, meaning the result *R* is prepared to send back to the activity *s* of service *f*. This tuple in the service solution will then activate a pair of rules to perform message passing between services.

```
let reply =
    replace "RESULT":<?R>, "CLIENT":s:f
    by "REPLY":s:f:<R>
in...
```

Fig. 9. Chemical rules for "Reply" activity

Structure activities form the "skeleton" of a BPEL process. A structural activity can be seen as a container to hold several basic activities. All the basic activities have to be executed according to the special execution logic defined by this "container", such as "sequence" expresses sequential execution while "flow" specifies parallel execution. In chemical computing framework, structural activities are performed by a set of chemical rules.

Sequence. On "workflow" level, each sub-solution represents a workflow component. As a result, to express the execution logic, a "pointer" has to be assigned to each component indicating next execution step(s). Each *component sub-solution* contains a "NEXT":< step_names > tuple. The second fields employs a multi-set to contain the names of next execution steps. In the framework shown in Figure 5, if "Inv_1" and "Inv_2" have to be executed sequentially, "Inv_1" solution must contain a tuple like "NEXT":<"Inv_2">.

This tuple will react with "*nextStep*" rule, its definition is given out in Figure 10. Once a component *s* has finished its calculation, a "RESULT" tuple is generated in its operation solution (if this is an invoke activity, this "RESULT" tuple can be received from a partner service). Meanwhile, *s* specifies its next execution step as *t* and the sub-solution of *t* is inert at that moment, this rule will encapsulate the result *r* into an "input" tuple and pass it to the solution of *t*. In this way, sequential execution is performed.

```
let nextStep =
    replace t:<?l>,
      s:<"NEXT":<t,?w>, "RESULT":<?r>>
    by s:<"NEXT":<w>, "RESULT":<r>>
       t:<l,"INPUT":<r>>
in...
```

Fig. 10. Chemical rules for structure activity

Parallel. Using the same way, it is easy to express parallel execution. Suppose that after activity "Inv_1", "Inv_2" and "Inv_3" will be executed in parallel, the operation solution of "Inv_1" has to contain the following tuple: *"NEXT":<"*Inv_2*","*Inv_3*">*. In this way, rule *nextStep* will pass the result of "Inv_1" to both "Inv_2" and "Inv_3" solutions simultaneously in order to start the invocations in parallel.

Throw. As error is inevitable during the execution of a program, especially for such complex distributed applications as Web service. WS-BPEL employs "throw" activity to alarm BPEL engine to take measures when error happens. In chemical computing, error checking is performed by rules. These rules plays the role of a watchdog that monitor the system in a certain level. Once there is an error detected, a "FAULT" tuple will be generated in the form of "FAULT":<fault info>, which encapsulates the detail of an error. This tuple will activate respective rules which implement fault handler to treat with the errors.

Fault Handler. A fault handler defines the response to a certain type of fault. In chemical model, it is also implemented by chemical rules. Once a "FAULT" tuple is thrown in a certain level, the respective fault handler has to take actions to deal with the error. For example, if a service calculates the maximum integer number, it requires a set of integers as inputs. Once a chemical rule *checkInput* detects that some float numbers are passed into this service solution as a parameter for

this invocation, a tuple "FAULT":<"Require integer numbers, get string input"> will be generated. This tuple will activate the *FHTypeError* rule which encapsulates this fault tuple in a "REPLY" tuple and then send it back to the invoker.

C. Service Level Coordination

In chemical framework, the interaction between services is expressed by writing message molecules into the target *service solution*. These molecules can react with some chemical rules in the partner *service solution* and the computation starts. This interaction is performed by a set of rules defined in the container.

As defined in Figure 11, rule *withdrawServiceCall* extracts the "CALL" tuple from the calling service's solution (expressed by "f:< >") to the *container*. This operation is from *workflow level* to *service level*. And then, rule *depositeServiceCall* moves the "CALL" tuple from container to the target service solution (expressed by "t:< >"). This operation is from *service level* to *workflow level*.

```
let withdrawServiceCall =
    replace f:<"CALL":s:t:<?w>,?l>
    by f:<l>, "CALL":s:f:t:<w>
in
let depositeServiceCall =
    replace t:<?w> "CALL":s:f:t:<?l>
    by t:<w, "CALL":s:f:t:<l>>
in...
```

Fig. 11. Chemical rules for "invoke" activity

The "reply" message is performed in the liked manner. As defined in *Figure 12*, rule *withdrawServiceReply* extracts the "REPLY" tuple from a partner service's solution to the container while *depositeServiceReply* passes it into the target service solution.

```
let withdrawServiceReply =
    replace f:<"REPLY":s:t:<?w>,?l>
    by f:<l>, "CALL":s:t:<w>
in
let depositeServiceReply =
    replace t:<?w> "REPLY":s:t:<?l>
    by t:<w, "REPLY":s:<l>>
in...
```

Fig. 12. Chemical rules for "Reply" activity

Based on the framework proposed in Section III-A, Figure 13 organizes all the rules introduced above into three hierarchical levels of solutions. This enriched framework can implement most of WS-BPEL constructs to express workflow and perform service orchestration.

IV. Implementation

This is the first study on investigating the possibility to use chemical programming model on service orchestration. As a result, a simple prototype is developed by implementing the chemical framework presented in the previous section. This prototype aims at proving the feasibility of using HOCL as a "glue" between services. Therefore, instead of using complex business services to provide commercial functionality, some basic calculating services are implemented. These services perform certain mathematical computing and all of them are developed in HOCL. More elaborate and advanced implementation will be carried out in the near future.

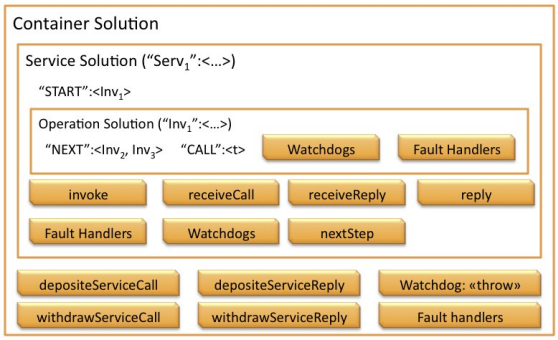

Fig. 13. Organization of rules

The context is shown in *Figure 14*, to calculate the maximum and minimum prime numbers included in a set of integers, a composite service named "*MaxMinPrimeNumbers*" is built by coordinating three partner services. The dashed arrows between the "*operation solutions*" in the composite service form the workflow. Firstly, it invokes "*Prime*" partner service to get all the prime numbers and in the following steps, it calls respectively "*Max*" and "*Min*" partner services to get the maximum and minimum value. Those two steps are carried out in parallel. On receiving the results from both steps, a reply message is generated in the final step.

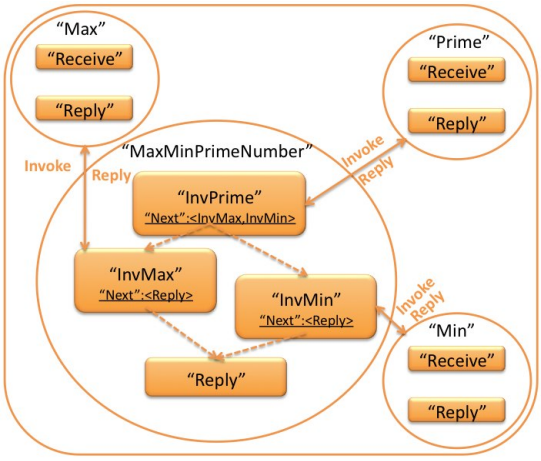

Fig. 14. Implementation using chemical model

An HOCL program creates a multi-set acting as the container. Each afore-mentioned service is represented by a *ser-*

vice solution in the container. Executing this program, the whole system is inert at the beginning, no chemical rule is activated. Imagine that if there is a "*client*" service which invokes "*MaxMinPrimeNumbers*" service, it has to pass a "CALL" tuple from its *service solution* to the container. This tuple will give rise to a series of chemical reactions. We do not actually implement a "client" service (there is no "client" *service solution* in the container); instead, we use the "shell" to add the following "CALL" tuple directly into the container to simulate an invocation from a "client" service:

```
"CALL":"Inv1":"client":"MaxMinPrimeNumber"
    :<16,7,8,0,12,3,4,5,7,10>
```

A shell is a monitoring tool with user interfaces for managing the container. It provides several commands which operate on the multi-set. For example, by using the shell, a user can either print out the container states (its current content) or add/remove molecules [11]. This "CALL" tuple implies that the activity "Inv_1" of service "*client*" invokes service "*MaxMinPrimeNumber*" with certain parameters. Once it is presented in the container, the rule "*depositeServiceCall*" will be activated to forward parameters into the *service solution* of "*MaxMinPrimeNumbers*". In this way, the computation in the composite service starts and it will invoke other partner services in succession. Finally, the following "REPLY" tuple is generated in the container which encapsulates the final result:

```
"REPLY":"Inv1":"client":<7,3>
```

As we have not implemented the "client" *service solution*, this "REPLY" tuple will be finally left in the container for us to check the computing result. It tells that the result <7,3> is for the "Inv_1" activity of "client" service. We can see that the expecting result is obtained and the whole process is automated.

In addition, a fault handler is implemented in the workflow level of "*MaxMinPrimeNumbers*" service in case of illegal input parameters. To check its validation, add another "CALL" tuple defined as follows into the container through the shell:

```
"CALL":"Inv1":"client":"MaxMinPrimeNumber"
    :<1,1.16,3,21,15,13,19,17,18,16>,
```

For the composite service, the valid input parameters have to be uniquely integers. As a result, the float number 1.16 will be detected as an illegal input, a certain rule can detect this error and generate a "FAULT" tuple with detailed information. Once a "FAULT" tuple is presented in its *service solution*, the fault handler will then stop the execution of the workflow and reply to its invoker with error information. Finally, the following "REPLY" tuple is left in the container:

```
"REPLY":"Inv1":"client":<"FAULT":<"Require
 integers, but get float parameter(s)!">>
```

Compare to the static nature of WS-BPEL, our approach is more dynamic. The definition of a business process can be modified on-the-fly. As shown in *Figure 14*, if the "Min" service has to be adapted, it is possible to refer to another partner service by replacing the "CALL" tuple in "InvMin" sub-solution. By using the "shell", the previous tuple "CALL":"Min" can be removed and a new tuple "CALL":"MinNumber" is added which is used to invoke "MinNumber" service that provides the same functionalities as "Min". In this way, the real time service adaptation can be performed. In the following work, we are going to implement service adaptation based on different strategies.

V. Related Works

The most common way to build composite applications is to use WS-BPEL. One of the drawbacks of WS-BPEL is its lack of formal semantics. [5], [12] have addressed this problem. The authors have formalized a novel service orchestration language: Webπ_∞, extending from π-calculus [13]. This language can be seen as a simplification of WS-BPEL. It defines the semantics to WS-BPEL, but the current work is limited in a subset of the whole WS-BPEL, such as basic activities, structured activities and error handling. On the other hand, WS-BPEL lacks monitoring and dynamic (runtime) adaptation mechanisms, a system "VieDAME" is proposed in [14] allowing monitor BPEL processes and replace existing partner services based on various replacement strategies.

Besides this traditional method, some novel approaches were proposed. An unconventional approach known as "tuple-space based Web service orchestration" implements a Linda-tuple [15] liked space to orchestrate web services. This viewpoint shares some similarities with chemical programming paradigm. [16] presents an infrastructure and a set of tools named TSpaces Services Suite (TSSuite) for the development and management of Web services. It extends the tuple space model to handle Web services as first class citizens. Each service manages a tuple space; If there are some requesting tuples in the its space, the service will perform the promised operation. In this way, the workflow is divided up to several sub-workflows, which are distributed in the internal or external region of an association. This has greatly enhanced the fault-tolerant and parallel access.

In [17], authors present xSpace, a tuple space that deals with XML documents natively and distributed across the internet. It can serve as a vehicle to orchestrate web services by providing an asynchronous interaction paradigm. The workflow controller uses xSpace to put in requests for the next workflow task and reads the results from xSpace before deciding the next executing step. The performance can be improved by integrating a notification system.

Recently, chemical programming is regarded as a suitable candidate for service coordination. [18] investigates the possibility to apply chemical programming on service orchestration. Some rules are defined to perform message passing among services. This work can be integrated into the framework proposed in this paper, the interaction between service solutions is implemented by a set of rules in service level. But [18] does not address any idea on how to express workflow using chemical programming model.

[19] provides a coordination framework based on chemical computing paradigm for advanced workow enactment. The authors proves that most of workflow constructs can be expressed by γ-calculus. γ-calculus is a formal definition of the chemical paradigm from which all these chemical models can be derived. But this work has a limitation in express workflow. For example, it use higher-order property to express sequential execution. That means if you have a thousand sequential execution steps, you have to compose a nesting solution structure with a thousand levels of nesting sub-solutions. Our framework provides structural templates for all kinds of workflow constructs.

VI. Conclusion

Nowadays, the hunger for designing loosely-coupled distributed applications requires service collaboration across organization boundaries. These applications are based on the composition of a set of independent software modules that are spread over computer networks. Service-Oriented Architecture brings us standards-based, robust and interoperable solutions. The most common way to build composite applications is to use WS-BPEL.

In this document we have introduced an unconventional approach: using chemical computing model to perform the orchestration of a large number of services for applications that require self-adaptation and fault-tolerance. A framework is proposed to define workflow and express service orchestration using chemical concepts. In this framework, services are represented by solutions and the orchestration is performed by a set of rules. We have shown that most of WS-BPEL constructs can be expressed using HOCL. As a proof of concept, a prototype is developed in HOCL. A composite service is created by coordinating the invocations to three predefined partner services and finally the expecting results are obtained.

Compare to the other traditional methods, our approach has following superiorities: first and foremost, it is highly dynamic. A service definition can be modified without stopping the execution of workflow; furthermore, it gets its semantic from chemical metaphor, which enables us to reason a process behavior; thirdly, the implementation of multi-set can be distributed that makes the application more scalable and reliable; finally, by using our chemical framework, an ongoing research for a GUI based code generator will make users feel free to deal with complex and error-prone XML-based grammar. All these advantages make us believe that chemical programming is a competitive solution for service orchestration programming.

This chemical framework can be improved in many aspects. Currently, the container solution is implemented in an HOCL program. All the services are represented by the sub-solutions in the container. This highly centralized implementation is opposed to our intention. We are going to distribute the implementation of the chemical framework. Each HOCL program creates a multi-set representing a *service solution*. These HOCL programs implement the *container solution* in a distributed way. Furthermore, in this prototype, all the services are developed in HOCL. However, in reality, a service can be written in various programming languages. In the next step, we are going to establish the interaction between our chemical framework with the real Web services. At that moment, each *service solution* acts as a representative of a real service (concrete service) in the chemical level. The chemical framework works like a middleware which is designed for service adaptation.

Acknowledgment

The research leading to these results has received funding from the European Communitys Seventh Framework Programme [FP7/2007-2013] under grant agreement 215483 (S-CUBE).

References

[1] Y. Vasiliev, *SOA and WS-BPEL: Composing Service-Oriented Solutions with PHP and ActiveBPEL*. Packt Publishing, September 2007.

[2] F. Daniel and B. Pernici, "Insights into web service orchestration and choreography," *INTERNATIONAL JOURNAL OF E BUSINESS RESEARCH*, vol. 2, no. 1, pp. 58–77, 2006.

[3] G. Alonso *et al.*, *Web Services - Concepts, Architectures and Applications*. Springer Verlag, 2004.

[4] *Web Services Business Process Execution Language*, OASIS Standard, April 2007.

[5] R. Lucchia and M. Mazzara, "A pi-calculus based semantics for ws-bpel," *Journal of Logic and Algebraic Programming*, pp. 96–118, January 2007.

[6] J. P. Banâtre, P. Fradet, and Y.Radenac, "Principles of chemical programming," *Electronic Notes in Theoretical Computer Science*, vol. 124, pp. 133–147, March 2005.

[7] J.-P. Banâtre, P. Fradet, and Y. Radenac, "The chemical reaction model recent developments and prospects," *Software-Intensive Systems and New Computing Paradigms: Challenges and Visions*, pp. 209 – 234, 2008.

[8] J.-P. Banâtre and D. Le Métayer, "Programming by multiset transformation," *Commun. ACM*, vol. 36, no. 1, pp. 36 98–111, 1993.

[9] J.-P. Banâtre, P. Frade, and D. L. Métayer, "Gamma and the chemical reaction model: Fifteen years after," *Multiset processing*, pp. 17–44, 2001.

[10] J. P. Banâtre, P. Fradet, and Y.Radenac, "Generalized multisets for chemical programming," *Mathematical Structures in Computer Science*, vol. 16, pp. 557 – 580, August 2006.

[11] C. Wang and T. Priol, *HOCL Programming Guide*, INRIA Rennes, September 2009.

[12] M. Mazzara and S. Govoni, *A Case Study of Web Services Orchestration*. Springer Berlin / Heidelberg, May 2005, vol. 3454/2005.

[13] D. Sangiorgi and D. Walker, "The pi-calculus: a theory of mobile processes," *Cambridge University Press*, 2001.

[14] O. Moser, F. Rosenberg, and S. Dustdar, "Non-intrusive monitoring and service adaptation for ws-bpel," *International World Wide Web Conference, Proceeding of the 17th international conference on World Wide Web*, pp. 815–824, 2008.

[15] S. Ahuja, N. Carriero, and D. Gelernter, "Linda and friends," *Computer*, vol. 19, pp. 26–34, August 1986.

[16] M. Fontoura, T. Lehman, D. Nelson, and T. Truong, "Tspaces services suite: Automating the development and management of web services," *In Proceedings of the 12th International World Wide Web Conference*, 2003.

[17] B. Umesh and B. Siddharth, "xspace: a tuple space for xml & its application in orchestration of web services," *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pp. 766–772, 2006.

[18] J.-P. Banâtre, T. Priol, and Y. Radenac, "Service orchestration using the chemical metaphor," *Software Technologies for Embedded and Ubiquitous Systems*, vol. 5287, pp. 79–89, September 2008.

[19] Z. Nemeth, C. Perez, and T. Priol, "Workflow enactment based on a chemical metaphor," *Software Engineering and Formal Methods*, pp. 127– 136, September 2005.