Formal Specification of Adaptive Web Service Composition Using Maude

Yasser Oussalah, Nadia Zeghib
LIRE laboratory
Mentouri Constantine University
Constantine, Algeria
Oussalah.yasser@hotmail.fr, n_zeghib@hotmail.com

ABSTRACT: *The composition of heterogeneous Web services is a key aspect of usability and applicability of Web services in different application domains such as business applications, healthcare, and e-government. Unfortunately, the reuse of Web services raises composition issues since they present, most of the time, mismatching at different levels such as structural, behavioral and non-functional. The resulted mismatches require adaptation to insure the correct working between Web services. This paper presents an effective adaptive Web service composition. It gives especially a formal specification of operations which enable to compose and adapt services at runtime. This specification is given in Maude language as a logical framework. Thanks to many Maude tools, formal analysis may be easily performed.*

Keywords: Web Service Adaptation, Dynamic Composition, Formal Specification, Maude

Received: 3 May 2012, Revised 5 June 2012, Accepted 9 June 2012

1. Introduction

Last years have seen the emergence of the service oriented architecture (SOA) designed to facilitate the creation, the publication, the networking and the reuse of applications based on services. Web services are the most important realization of the SOA architecture. They are self-contained, self-describing, modular applications that can be published, located, and invoked across the Web. Nowadays, an increasing amount of companies and organizations only implement their core business and outsource other application services over Internet. Thus, the ability to efficiently and effectively select and integrate inter-organizational and heterogeneous services on the Web at runtime is an important step towards the development of the Web service applications.

Web service composition has been emerged as an important strategy to allow enterprise collaboration [1]. Moreover, Web services composition simplifies the rapid development of applications by enabling the reuse of existing components. The composition problem appears when a client's request could not be satisfied by any of the existing services on its own. Hence, it becomes necessary to combine them in order to create a new value-added composite Web service. Particularly, the *dynamic* Web service composition is very promoting because it enables the user to select, at runtime, existing Web services to provide an unlimited number of new services from limited set of services. This dynamic feature of service composition provides flexibility and adaptability to applications. For example, an application built on top of the dynamic service composition system is able to change its user interface dynamically according to user's preference (e.g. English/Japanese menu, colorful/simple buttons,

…etc.), Furthermore, a totally new application may emerge by combining several components designed for entirely different purposes.

In the research related to Web services, several initiatives have been conducted with the intention to provide platforms and languages that will allow easy collaboration, composition and integration of heterogeneous systems. In particular, some standards have been developed such as Universal Description, Discovery, and Integration (UDDI) [2], Web Services Description Language (WSDL) [3], Simple Object Access Protocol (SOAP) [4], Business Process Execution Language for Web Services (BPEL4WS) [5].

Despite all these efforts, the Web service composition still is a highly complex task. One source of this complexity is the mismatches that may occur between two services in the composition process. In fact, the composition of Web services involves wiring together multiple Web services and having them interact often in ways not originally foreseen during their initial development. In doing so, it is unavoidable that mismatches may arise at different levels: signature, behavior, quality of service and semantics. Hence, there is a need for adaptation method to overcome these mismatches without modifying the service code due to its black-box nature. The adaptation ensures correct working and interaction among the involved components in the composition.

This paper presents a formal specification of dynamic Web service composition and adaptation [6]. For this purpose, we use Maude language to define basic adaptation operations and to specify the dynamic composition process.

The rest of this paper is organized as follows. Section II presents an overview on Maude system. Section III gives specification of the Web service interface. Section IV presents the specification of Web service compatibility. Section V details the adaptation operations. Section VI gives the specification of adaptive composition. Section VII discusses related work and existing approaches. Finally, last section concludes the paper with future work.

2. MAUDE overview

Maude [7] [8] is a high-performance language and system supporting both equational and rewriting logic specification and programming for a wide range of systems and applications. Equational theories describe the *static* parts of a system and are represented as *functional modules*. Rewrite theories describe the *dynamic* parts of the system and are represented in Maude as *system modules*.

A functional module specifies one or more data types and operations on them. The sorts of the data types are declared with the keyword *sort*, and subsorts are specified using *subsort*. Functions are declared by *op* declarations of the form

$$\textbf{op}\ f : s1 \ldots sn \rightarrow s\ [\text{attributes}].$$

Where f is a function symbol and $s1,\ldots, sn$ and s are sorts. If the number of arguments of f is zero, then f is called a *constant* of sort s. Function symbols can be declared with both prefix and "mix-fix" form. In the latter case, the positions of the arguments are given by underscores ('_') in the function declaration. A function declaration may also contain attributes that specify properties of the function. For instance the attributes [comm][assoc][ctor] designate respectively that the operation is commutative, associative and constructor. A functional module can contain equations and variable declarations. Variables are either declared separately with the keyword *var*, or within the equations. Equations may be either unconditional or conditional:

$$\textbf{eq}\ t = u\ .$$

$$\textbf{ceq}\ t = u\ \text{if}\ cond$$

System modules can, in addition to the declarations described in a functional module, contain unconditional and/or conditional *rewrite rules* of the form

$$\textbf{rl}\ [label]: t \Rightarrow u\ .$$

$$\textbf{crl}\ [label]: t \Rightarrow u\ \text{if}\ cond$$

Computationally, a rewrite rule of the form $t \Rightarrow u$ describes an atomic and local transition in the system. *Logically* it is an inference step from sentences of type t to sentences of type u.

Although there exist many formal models like finite state machines [21] [22], Petri nets [23] [25], and labeled transition systems [24], to specify the Web services and their composition, Maude seems adequate to do, since it is based on rewriting logic which is a unified framework for all these concurrency models. Moreover, Maude supports in a systematic and efficient way logical reflection. This makes Maude remarkably extensible and powerful, supports an extensible algebra of module composition operations, and allows many advanced metaprogramming and metalanguage applications. Indeed, some of the most interesting applications of Maude are metalanguage applications, in which Maude is used to create executable environments for different logics, theorem provers, languages, and models of computation. Furthermore, Maude specifications are executable, and can be subjected to simulation and formal analysis using the Maude interpreter.

3. Specification of Web service Interface

Web services are modular applications which can be only viewed through their interface definition due to their black box nature. The aim of this section is to specify formally the components of the Web service interface.

3.1 Messages Specification

The cooperation between Web services is based on message exchange. These messages are either input or output ones. Each Web service operation consumes some incoming messages and/or produces some outcoming ones. The Maude specification of messages is the following.

```
Sorts MesageId MessageIN MessageOUT Messages.
Sorts ListMId DirectionIN DirectionOUT.
op msgs(_,_): MessageIN MessageOUT -> Messages.
op msgIN(_,_):ListMId DirectionIN -> MessageIN [ctor].
op msgOUT(_,_): ListMId DirectionOUT -> MessageOUT
[ctor].
op SM : -> MessageId [ctor].
op TM : -> MessageId [ctor].
```

ListMId is a list of message identifiers. *DirectionIN* (resp. *DirectionOUT*) is used if the message is an input (resp. an output) one. The *MessageId* can have two values either the source message *SM* or the target message *TM*.

3.2 Operations Specification

A Web service may provide many functionalities, each of them implemented by an operation. Hence, a Web service can be viewed as a set of operations. An operation is defined by its name, its input and output message types, i.e. *o*: = < *name, data Input, data Output* >.

Messages need to be grouped into operations, which may define an < input >, and an < output > message [9].

The operations specification is given below.

```
op Op(_,_) : OperationId Messages -> Operation [ctor].
```

An operation identifier (*OperationId*) can be either source operation (*SO*) or target operation (*TO*) according to their occurrence in the composition:

```
op SO : -> OperationId [ctor].
op TO : -> OperationId [ctor].
```

3.3 Interface specification

A Web service interface is defined by one or more operations. For the sake of simplicity we attribute to each service one operation to shorten the specification process of composition. The constructor of a Web service interface is the following Maude operation.

```
op Inter : Operation -> Interface [ctor].
```

4. Specification of Web service Compatibility

Interactions between Web services involve the exchange of messages. Hence it is important to check that the data types and number of the message parameters sent by a service are compatible with the parameters required by its partner. This requires pre-conditions of inputs and post-conditions of the outputs. In fact the composition of two Web services requires finding two compensable operations (one of each) that can be linked: two operations can be linked when the output parameters of the first (source) can cover the input parameters of the second (target).

When two services WS1, WS2 have message types respectively *SM*, *TM*, which means that the sent messages (source messages *SM*) from the first service are received as target messages in second service (output1= *SM* and input2 = *TM*) , and the messages from both have the same cardinality, so there is a compatibility between the messages provided and required (output1, input2). Otherwise there is a need for adaptation.

The operations of the services WS1 and WS2 must have respectively identifiers: *SO* and *TO*. This means that the operation *SO* of the first one has its related operation *TO* in the second one (*SO* and *TO* are compensable operations to perform an adequate Web service composition).

To insure the compatibility between two services, we have to check if the sent messages are compatible with the required ones, and each operation must be linked to its related one.

The operation *check-compatibility* is specified as follows:

```
op check-compatibility: Interface Interface-> Bool
eq check-compatibility(I,I2) = if getOId(I) == SO
and getOId(I2) == TO
and  lenght(getLMOUT(I)) == lenght(getLMIN(I2))
and  verifyType(getLMOUT(I), getLMIN(I2)) == true
then true else false fi.
```

The operations *getOId*, *getLMIN* and *getLMOUT* used in the above equation allow to get respectively the identifier of the operation (*SO/TO*), the list of input messages, and the list of output messages. The operation *verifytype* checks the compatibility between messages types.

5. Specification of Adaptation operations

The scenarios in the Figure 1 show possible mismatches which may occur at runtime between the involved components in the composition. In the first scenario (*a*) the Web service sends two different messages (*A* and *B*) while only one of them (*A*) is expected. In the scenario (*b*) messages are sent aggregated (*A*+*B*) when they are needed to be separated. The third scenario (*c*) is the reverse of (*b*): the messages are sent separately when they are needed in aggregation. In the last scenario (*d*), the type of the sent message does not match with the required type.

To insure the correct working among the involved components in the service composition we propose a set of adaptation operations which perform the needed mapping of interfaces.

We have used in the process of adaptation four operations to perform the needed mapping. Three of them are presented in [10], which helps to fulfill the compatibility requirement these operations perform the following functions:

- ***Collapse*:** is used when a stream of messages is aggregated into a single message.
- ***Burst*:** works in the reverse of the Collapse operation and it is used when a single message needs to be split into a stream of messages.
- ***Hide*:** is used when a message from the source interface is not required in the target interface.
- ***ResvType*:** is used when the type of the message provided in the source interface is not compatible with the required one in the target interface.

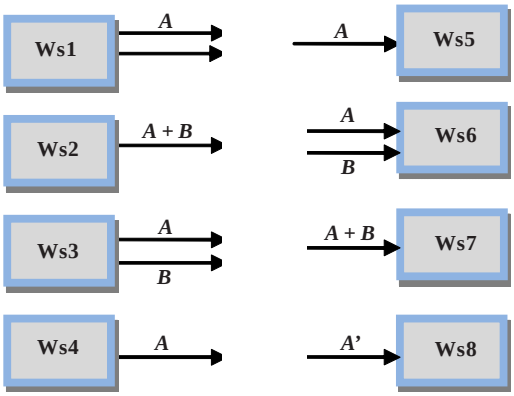

Figure 1. Mismatch scenarios

The profile of each adaptation operation is specified in Maude by the following Maude operations.

```
op collapse: ListMId -> MessageId.
op burst: MessageId  Int -> ListMId.
op hide: Int ListMId -> ListMId.
op resvType: ListMId -> ListMId.
```

The sort *ListMid* stands for list of messages identifier (*SM* or *TM*). The formal semantics of these operations is specified by the following set of Maude equations.

```
eq collapse(MI: LMI) = MI.
eq burst(MI, 1) = MI: nil.
eq burst(MI, N) = MI: burst(MI, N - 1).
eq hide(1, MI: LMI) = LMI.
eq hide(N, MI: LMI) = MI: hide(N - 1, LMI).
eq  resvType(MI: nil) = SM: nil.
ceq resvType(MI: LMI) = SM: resvType(LMI)
      if MI == TM.
eq resvType(MI: LMI) = MI: resvType(LMI)[owise].
```

6. Specification of Adaptive composition

When a Web service is unable to provide alone a user request, it communicates with other Web services either to provide a part of the requested service or to request another part of it. This is known as Web service composition.

A composite Web service consists of several conceptually autonomous but cooperating units in order to establish a long-running service composition [11]. Consequently, each operation involved in the composition process will be contained in the functionalities of the composite service(CWS):

$$op \times op \cdots \times op \text{ -> } CWS$$

The composition of *n* Web services is defined recursively by the function Rec_comp as follows:

- Rec_comp (n) = compose (Rec_comp (n-1), Ws_n) if $n \geq 2$.
- Rec_comp (1) = Ws1 .

For instance: Rec_comp (2) = compose (Rec_comp (1), ws_2)
= compose (ws1, ws2) .

In fact, the composition process is highly complex task since it requires the ability to discover or detect pairs of services such

that the output of one service is equal or equivalent to the input of another (correspondence between interfaces). If the mismatch occurs, the adaptation will be performed using the mapping operations presented previously. Hence, we resolve inadequacy resulted due to number and type of parameters.

According to the user request, our approach selects at runtime the adequate services (Ws_1,Ws_2,….,Ws_n), verifies the compatibility and composes them if no mismatch occurs. Otherwise the adaptation is required using mapping operations .

We consider *listInter* a list of selected services interfaces for composition as follows:

```
eq listinter(LI) = Inter(Op(SO, msgs(msgIN(TM:
nil, IN, msgOUT( SM: TM: TM: SM: nil, OUT)))) ++
Inter(Op(To, msgs(msgIN( TM: TM:
nil,IN, msgOUT(SM: nil, OUT)))) ++ nil.
```

The operation *compose* defined below, composes the services if there is compatibility between them. In the case of mismatch, the operation *resolves-mismatch* will be invoked until the interfaces become compatible to compose the services.

```
op compose :Interface Interface -> ListInter .
 eq compose(I,I2) = if check-compatibility(I,I2) ==
true then comp(I,I2) else comp(resolve-
mismatch(I,I2,false),I2) fi .
```

The operation *resolve-mismatch* (*I*, *I*2, *B*), adapts interface *I* to *I*2 when *B* = *false* using the adaptation operations presented previously.

```
op resolve-mismatch: Interface Interface Bool -> Interface.
```

Depending on the case of mismatch, the operation *resolve-mismatch* invokes one of the following operations :

• **collapse:** when stream of messages need to be aggregated into one single message (ie: the length of output messages list is ≥ 2 and length of required target messages list is equal 1).

```
ceq resolve-mismatch(I,I2,B) = resolve-
mismatch(setLMOUT(collapse(getLMOUT(I)):
nil,I),I2,check-
compatibility(setLMOUT(collapse(getLMOUT(I))
: nil,I),I2))
     If lenght (getLMOUT (I)) >= 2 /\ lenght
       (getLMIN (I2)) == 1 /\ B == false.
```

• **burst:** when a single message needs to be split into a stream of messages (ie: length of output messages list = 1 and the length of target input messages list ≥ 2).

```
ceq resolve-mismatch(I,I2,B) = resolve-
mismatch(setLMOUT(burst(elmAt(1,getLMOUT(I),
lenght(getLMIN(I2))), I),I2, check-
compatibility(setLMOUT(burst(elmAt(1,getLMOUT(I)),
lenght(getLMIN(I2))), I), I2))
       if lenght(getLMOUT(I)) == 1 /\
       lenght(getLMIN(I2)) > = 2 and B == false.
```

• ***hide*:** when additional message need to be hidden because it's not required in the target interface (ie: when the length of ouput messages list is greater than the length of target input message list).

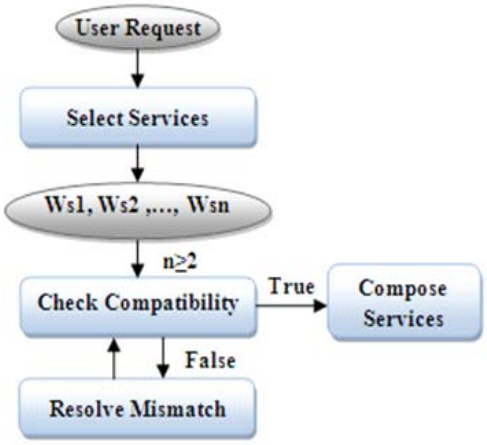

Figure 2. Main Steps of our approach for web service composition and adaptation

```
ceq resolve-mismatch(I,I2,B) = resolve-
mismatch(setLMOUT(hide(lenght(getLMOUT(I)),
getLMOUT(I)),I),I2,check-
compatibility(setLMOUT(hide(lenght(getLMOUT(I)),getLMOUT(I)),I),I2))
    if sus(getLMOUT(I),getLMIN(I2)) > = 1 /\ B
 == false.
```

• ***resvType*:** when the type of the message provided in the source interface is not compatible with the required one in the target interface.

```
ceq resolve-mismatch(I,I2,B) = resolve-
mismatch(setLMOUT(resvType(getLMOUT(I)),I),I2,check-
compatibility(setLMOUT(resvType(getLMOUT(I)),I),I2))
    if verifyType(getLMOUT(I),getLMIN(I2)) == false
    and B == false .
```

When the compatibility becomes true the operation *resolve- mismatch* will return the same adapted interface.

```
ceq resolve-mismatch(I,I2,B) = I if B == true.
```

Illustrating examples :
Let's treat some simple examples using the proposed specification in Maude.

```
eq listinter(LI) = Inter(Op(SO,msgs(msgIN(TM:
nil,IN), msgOUT( TM: TM:  nil, OUT)))) ++
Inter(Op(TO, msgs(msgIN( TM: TM:
nil,IN),msgOUT(SM: nil,OUT)))) ++ nil.
```

The selected Web services are found in the *ListInter* (*LI*) which is a list of linked interfaces of the form *I* ++ *I2* +…..+ *nil*. We can access to an element from the list of interfaces through the operation *elmAt* (*N*).

In the first example we treat the case when the mismatch raises because the type of provided messages is not compatible with required type of the target interface .

To compose the services in *Listinter* (*LI*) we invoke the operation *compose* (*elmAt* (1,*listinter*(*LI*)), *elmAt* (2, *listinter* (*LI*)))
The execution result of compose operation is given below.

```
Maude> select interlist .
Maude> red compose(elmAt(1,listinter(LI)),elmAt(2,listinter(LI))) .
reduce in interlist : compose(elmAt(1, listinter(LI)), elmAt(2, listinter(LI)))
    .
rewrites: 141 in -152581613208ms cpu (0ms real) (~ rewrites/second)
result ListInter: comp(Inter(Op(SO,msgs(msgIN(TM : (nil).ListMId,IN),msgOUT(SM
    : SM : (nil).ListMId,OUT)))), Inter(Op(TO,msgs(msgIN(TM : TM : (
    nil).ListMId,IN),msgOUT(SM : (nil).ListMId,OUT)))))
Maude>
```

Let's consider through the following example, an another case of mismatch when the provided messages (output messages) are sent in aggregation (length of output messages list = 1) when they are needed to be split (length of target input messages list = 2).

```
eq listinter(LI) = Inter(Op(SO,msgs(msgIN(TM:
nil,IN),msgOUT(SM: nil,OUT)))) ++ Inter(Op(TO,
msgs(msgIN(TM: TM:  nil,IN),msgOUT(SM:
nil,OUT)))) ++ nil.
```

```
Maude> select interlist .
Maude> red compose(elmAt(1,listinter(LI)),elmAt(2,listinter(LI))) .
reduce in interlist : compose(elmAt(1, listinter(LI)), elmAt(2, listinter(LI)))
    .
rewrites: 119 in -152599804981ms cpu (0ms real) (~ rewrites/second)
result ListInter: comp(Inter(Op(SO,msgs(msgIN(TM : (nil).ListMId,IN),msgOUT(SM
    : SM : (nil).ListMId,OUT)))), Inter(Op(TO,msgs(msgIN(TM : TM : (
    nil).ListMId,IN),msgOUT(SM : (nil).ListMId,OUT)))))
Maude>
```

The third scenario is the reverse of the second scenario, when the provided messages are sent split when they are needed in aggregation.

```
eq listinter(LI) = Inter(Op(SO,msgs(msgIN(TM :
nil,IN),msgOUT(  SM  : SM : SM : nil,OUT)))) ++
Inter(Op(TO ,msgs(msgIN( TM : nil,IN),msgOUT(SM:
nil,OUT)))) ++ nil .
```

```
Maude> select interlist .
Maude> red compose(elmAt(1,listinter(LI)),elmAt(2,listinter(LI))) .
reduce in interlist : compose(elmAt(1, listinter(LI)), elmAt(2, listinter(LI)))
    .
rewrites: 96 in -152587253129ms cpu (0ms real) (~ rewrites/second)
result ListInter: comp(Inter(Op(SO,msgs(msgIN(TM : (nil).ListMId,IN),msgOUT(SM
    : (nil).ListMId,OUT)))), Inter(Op(TO,msgs(msgIN(TM : (nil).ListMId,IN),
    msgOUT(SM : (nil).ListMId,OUT)))))
Maude>
```

In the forth scenario we treat the case of mismatch when an additional message in the provided messages is not required in the target interface.

```
eq listinter(LI) = Inter(Op(SO,msgs(msgIN(TM :
nil,IN),msgOUT(  SM : SM : SM : nil,OUT))) ++
Inter(Op(TO ,msgs(msgIN( TM : TM :
nil,IN),msgOUT(SM : nil,OUT))) ++ nil.
```

```
Maude> select interlist .
Maude> red compose(elmAt(1,listinter(LI)),elmAt(2,listinter(LI))) .
reduce in interlist : compose(elmAt(1, listinter(LI)), elmAt(2, listinter(LI)))
    .
rewrites: 153 in -152592820995ms cpu (0ms real) (~ rewrites/second)
result ListInter: comp(Inter(Op(SO,msgs(msgIN(TM : (nil).ListMId,IN),msgOUT(SM
    : SM : (nil).ListMId,OUT)))), Inter(Op(TO,msgs(msgIN(TM : TM : (
    nil).ListMId,IN),msgOUT(SM : (nil).ListMId,OUT)))))
Maude>
```

The above examples of Maude specification execution show that we achieved the compatibility in all cases using adaptation operations via the operation *resolve- mismatch*.

7 . Related work

Service interfaces can be described from a structural perspective (where the focus is on message types), and from a behavioral perspective (where the focus is on control dependencies between message exchanges). The problem of interface adaptation from the structural perspective has received considerable attention leading to a number of transformation definitions such as XSLT [12] and schema mapping tools such as Microsoft BizTalk Mapper [13], Stylus Studio XML Mapping Tools [14], and SAP XI Mapping Editor. However the problem of interface adaptation from behavioral perspective is still open. A number of studies in this field have been proposed. For instance, in [10] the authors describe the interface as ordered sequence of actions and they have proposed an algebra of transformation of interfaces, depending on the cases of mismatch that could occur to resolve inadequacy between interfaces. They take as input a source interface to produce a target interface by transforming the interfaces via six operators which are:

Flow : where a defined action in the source interface becomes another action in the target interface.

Gather: is applied when multiple actions from the source interface map to a single action in the target interface.

Scatter: is applied when a single action in the source interface is transformed into multiple actions in the target interface.

Collapse: is used when a stream of messages resulting from multiple instances of the same communication action is aggregated into a single message.

Burst: works in the reverse of the Collapse operator and is used when a single message needs to be split into a stream of messages.

Hide: is used when an action from the source interface is not required in the target interface.

In [15] the authors proposed an approach to the composition and adaptation of mismatching components in systems where the number of transactions is not known in advance. Their approach applies composition at run-time with respect to the composition specification, using π-calculus to specify component interfaces.

In [16] the authors specify mediator with finite state automata that resolves behavioral mismatches at runtime due to the removal of operations in provided interfaces, they also proposed an algorithm that resolves such mismatches .

In [9] the authors have proposed an approach for composition that only uses already available information in service interface

definitions. It does not require service providers to describe their interfaces with semantic markup. They proposed data types matching and service composition algorithm, using the measure of linguistic similarity between two data types.

In [17] the authors have identified a number of possible mismatches between services and some basic mapping functions that can be used to solve simple mismatches. Such mapping functions can be combined in a script to solve complex mismatches. Scripts can be executed by a mediator that receives an operation request, parses it, and eventually performs the needed adaptations.

In [18] the authors propose a process mediation architecture based on Triple space computing, and present potential solutions for resolvable message sequence mismatches. In addition, they categorize these resolvable mismatch scenarios into five classes. This analysis generalizes the resolvable message sequence mismatches, provides the basis for checking Web service compatibility from the behavioral aspect, and offers an opportunity to have a uniform solution to address these mismatches.

In [19] the authors present a framework for Dynamic service composition and parameters matchmaking. They discussed main problems faced by dynamic service composition. Among which are transactional support and compositional correctness. To make the system flexible they include user involvement at few steps for example selection of service and matchmaking decision.

In [20] the authors present a novel approach for formalizing Web service composition as an executable formal specification described in the Maude language Strategy, a recent extension of Maude. The formalization process is accomplished in two steps: (1) translating the BPEL description in an extension of UML 2.0 called UML-S "*UML for Services*" and (2) translating the UML-S graphical description generated to Maude's strategy language.

Our contribution regarding the most approaches is that we have used the dynamic composition and adaptation whereas the other approaches resolve either the dynamicity of the composition or the adaptation of static composition. In our previous work [6] we presented an algorithm that supports *both* dynamic composition and adaptation of Web services. The interface description of the services is used to detect the mismatches between services. The approach allows to perform the recovering of structural and behavioral mismatches via a set of mapping operations. We presented also a tool *CompAdapt* which implements the presented algorithm in Java language. In [6] we used simulation and testing process to validate our approach whereas in this current paper we present a formal approach using Maude to specify an adaptive Web service composition. Thus we endow the previous work [6] with formal aspects. Particularly, we provide a formal and executable specification in Maude which specifies the process of composition and adaptation automatically. The use of Maude system allows to perform formal verification and promotes the evolution of the composition specification by adding eventually new sorts, operations, equations and rules.

8. Conclusion

In this paper we presented a formal specification of dynamic web service composition and adaptation using Maude which is a high-performance reflective language and system, supporting both equational and rewriting logic specification. Maude supports in a systematic and efficient way logical reflection. Indeed Maude allowed performing an executable specification of web service composition and adaptation using a set of operations, all that based solely on interface description. Thanks to this logical framework we do not only obtain a high level specification of dynamic adaptation and service composition, but we are also able to execute it and formally reason on it.

In future it will be interesting to use the rewriting logic to simulate the process of web service composition and adaptation through rewriting rules and also specifying the function of composition for *n* elements recursively.

References

[1] Lins FAA, dos Santos JC, Rosa NS. (2007). Improving Transparent Adaptability in Web Service Composition, *In*: Proc. IEEE International Conference on Service-Oriented Computing and Applications, Newport Beach, CA, p. 80–87 .

[2] Bellwood, T. al. (2002). Universal Description, Discovery and Integration specification (UDDI), http://uddi.org/pubs/uddi-v3.00-published-20020719.htm.

[3] Chinnici, R. al. (2001). Web Services Description Language (WSDL), http://www.w3.org/TR/wsdl/.

[4] Box, D. al. Simple Object Access Protocol (SOAP), http://www.w3.org/TR/SOAP/.

[5] Andrews et al. T. (2003). Business Process Execution Language for Web Services(BPEL4WS), http://www106.ibm.com/developerworks/Webservices/library/ws-bpel, May.

[6] Oussalah, Y., Zeghib, N. An algorithm for Web service composition and adapatation based on interface description, *In*: Proceedings of ICITeS'2012, *International Conference on Information and e-Services*, Sousse,Tunisia.

[7] http://maude.cs.uiuc.edu/overview.html, [May, 2012].

[8] Lien, E. (2009 May 29) Formal modelling and analysis of the NORM multicast protocol using Real-Time Maude . Available: http://home.ifi.uio.no/peterol/RealTimeMaude/ NORM/thesis.pdf. [june 2012]

[9] Zhang, J., Yu, S., Ge, X., Wu, G. (2006). Automatic Web Service Composition Based on Service Interface Description, *In*: Proceedings of International Conference on Internet Computing. p. 120-126.

[10] Dumas, M., Spork, M., Wang, K. (2006). Adapt or Perish: Algebra and Visual Notation for Service Interface Adaptation, 4th *International Conference on Business Process Management*, 5-7 September, Vienna, Austria.

[11] Bao, L., Zhang, W., Xie, X. (2010). A Formal Model for Abstracting the Interaction of Web Services, *Journal of Computers*, 5 (1) 91-98, Jan.

[12] http://www.w3.org/TR/xslt

[13] http://msdn.microsoft.com/enus/library/ee253382(v=bts.10).aspx

[14] http://www.stylusstudio.com/xml_schema.html

[15] Camara, J., Salaun, G., Canal, C. (2007). Run-time Composition and Adaptation of Mismatching Behavioural Transactions, Fifth IEEE International Conference on Software Engineering and Formal Methods SEFM 2007, London, p. 381 - 390.

[16] Aït-Bachir, A., Fauvet, M. (2007). Reconciling Web Service Failing Interactions: Towards an Approach Based on Automatic Generation of Mediators, 16th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, WETICE, Paris.

[17] Cavallaro, L., Di Nitto, E. An Approach to Adapt Service Requests to Actual Service Interfaces, *In*: SEAMS '08 Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems. ACM, New York, NY, USA, p. 129–136.

[18] Zhou, Z., Sapkota, B., Cimpian, E., Foxvog, D., Vasiliu, L., Hauswirth, M., Yu, P. (2008). Process Mediation Based on Triple Space Computing. *In*: *Proceedings of the 10th Asia-Pacific Web Conference*, April, Shenyang, China.

[19] Allauddin, M., Azam, F. (2011). Dynamic Web Service Composition and Parameters Matchmaking *International Journal of Computer Applications* 36 (9) 21 - 26, December. *Published by Foundation of Computer Science*, New York, USA.

[20] Merouani, H., Mokhati, F., Seridi-Bouchelaghem, H. Towards formalizing Web service composition in Maude's strategy language, *In*: Proc. ISWSA, p. 15 - 15.

[21] Mohanty, H., Mulchandan, J., Chentahti, D., Shyamasundar, R. K. Modeling Web services with FMS modules, *In*: AMS '07 Proceedings of the First Asia International Conference on Modelling & Simulation, p. 100-105.

[22] Thirumaran, M., Dhavachelvan, P., Abarna, S., Lakshmi, P. (2011). Finite State Machine Based Evaluation Model for Web Service Reliability Analysis, *International Journal of Web & Semantic Technology* (IJWesT) October, 2 (4) ISSN: 0975 - 9026 (Online) 0976 - 2280 (Print).

[23] Caliz, E., Umapathy, K., Sánchez-Ruíz, A. J., Elfayoumy, S. A. (2011). Analyzing Web Service Choreography Specifications Using Colored Petri Nets, Service-Oriented Perspectives in Design Science Research, Lecture Notes in Computer Science, 6629, 412 - 426

[24] Pathak, J., Basu, S., Honavar, V. (2006). Modeling Web Service Composition using Symbolic*Transition Systems*, *In*: AAAI Workshop AI-SOC.

[25] Hamadi, R., Benatallah, B. (2003). A Petri Net-Based Model for Web Service Composition, *In*: Proceedings of the 14th Australasian Database Conference (ADC'03), CRPIT 17, p. 191-200, Australian Computer Society, Adelaide, Australia, February.