A Model-Based Validation Framework for Web Services

V. Pretre, F. Bouquet, C. Lang
Laboratoire d'Informatique de l'Université de Franche-Comté
16 route de Gray
25030 Besançon CEDEX
{pretre, bouquet, lang}@lifc.univ-fcomte.fr

Résumé

Nowadays, web services are more and more used for distant services. Since they are very present, there is an important need of validation and verification. As for any application, human error may occur during the development process.
In order to solve this problem, we propose in this paper a certification solution relying on model based testing. Results of tests are used to compute a mark that qualify quality of web services operations.
This solution is then integrating in a validation framework based on an UDDI server. In this framework, web services are tested when they declare to the UDDI server, and the obtained marks are supplied to customers seeking for services. In order to give concrete expression of this solution, we apply it to a blog publication example.

1 An introduction to web services quality

Web services (we note web service "WS", and web services "WSs") are a particular form of distributed software. As any other software, they are not fully reliable, their quality may change depending on several parameters (implementation, targeted computers, networks, ...). Thus, users can not be totally confident on results given by the WS. The aim of this paper is to solve a part of these problems by providing certification levels based on tests.

The first part of this paper presents WS context and why users may not be confident about these WSs. Then, we introduce solutions to solve this problem and at last, those solutions are merged into a validation framework.

1.1 Web services

The easiest definition for a WS is "server of a client-server scheme with communication based on XML messages" [13]. In fact the more commonly used protocol is http and messages are encapsulated with SOAP (Simple Object Access protocol).

The main interest of WSs is independence from proprietary technologies and programming languages.
Using XML to describe message content allows us to be independent from a common programming language for all members of the distributed application. Thus, we can easily describe exchanged datas without taking care about a given programming language. Moreover, a XML message can, when received, be transformed in a suitable form : class, record, ...
The use of well known protocols (http, https) makes WSs easy to deploy since many libraries exist to handle these protocols.

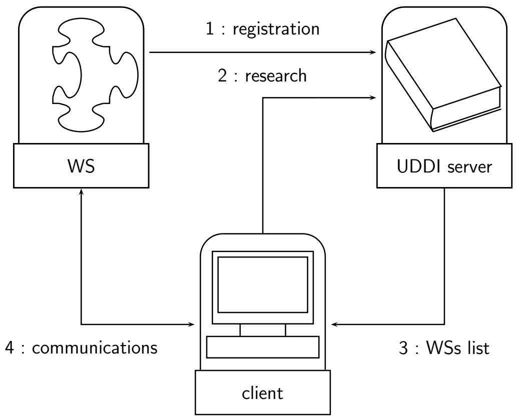

FIG. 1 – Life cycle of a WS

An important point in WSs is the automatic discovery, performed with UDDI servers [9]. When a WS wants to be discovered, it declares itself to an UDDI server using a WSDL file (Web Services Description Language) [6]. This file describes the different methods associated to the WS, how to call them, and the type of the result. When a WS (or any other software) needs another WS, it will request the UDDI server, who, depending on the demand, will give back a list of compliant possibilities and associated WSDL files.
Figure 1 shows the life cycle of a WS call using a UDDI server.

In this paper, we will use two WSs examples. The first one is a blog framework WS. It supplies ten operations : **register** (permits to an user to create an account on the framework), **login** (logs the user in the WS), **logout** (logs the user out of the WS), **create** (creates a new post), **save** (saves the current post), **prePublish** (prepares the post for publication), **tag** (add tags to the post), **categorize** (puts the post in categories), **publish** (publishes the post), **open** (open a previously saved post).
The second WS is a blog directory WS, which provides one operation : **ping**. This operation is used by blogs to inform the directory that a new post has been published. For this article, we do not consider the web interfaces of the blog framework and the blog directory.

As we can see in the blog framework, some operations need other one. We introduce now relations existing between operations.

1.2 Dependencies between operations

There exist two kinds of dependencies between operations : temporal dependencies and compositions which are introduced first.

Composition : a composition is the fact that an operation acts as a client of another one. In our example, the **publish** operation call the **ping** operation to inform the blog directory that a new post has been published.
In this article, we consider that compositions can be of two types : those where results produced by composed operation impact composing operation's behaviour, and those in which the result is not used (in our example, the success of the **ping** operation has no effect on the publication).

Temporal dependency : there is a temporal dependency when an operation can be called only if another operation has been called previously. In our example, it is impossible to open a post if no post has been saved previously.
Temporal dependencies can be split in two categories : **"at least once"** and **"each time"**. We consider two operations a and b, where a temporally depends of b. If the temporal dependency is of type "at least once", then we have to call b once, and then we can call a as much as we want (in our example, **login** and **register** have this kind of dependency).
If the dependency is of type "each time", if we want to call a, we must call b before, and this every time we want to call a (in our example, **publish** and **create** have an "each time" temporal dependency).

These two types of dependencies have an impact on WSs quality that we introduce now.

1.3 Web services quality

Confidence in WSs can be expressed in many ways : quality of results, computing time, treatment of private datas ...
In this paper, we focus on the first point : quality of results. It is based on two criteria : correctness and completness of results.

This quality depends on three parameters : development phase, network between client and WS and relations between operations. The development phase impact on quality result is obvious. WSs are, as any software, subjects to bugs due to bad implementation.
Even if the WS used is fully reliable, network can lead to quality problems. Results produced by the WS are right, but during transport some packets may be lost or corrupted.
The last source of quality problems is relations between WSs operations. In most compositions cases, the composing operation uses composed operation's results to compute its own ones. That leads to errors propagation if composed operations produces wrong results. Similar problems rise with temporal dependencies : the difference is that results are not directly shared from operation to operation, but passes through a third party (the customer or server side, in a database for example).

To solve this problem, we propose a solution based on WSs certification. First, we introduce the certification process, and then how we apply it in our validation framework.

2 The certification process

The certification process we propose is done in four steps :

– modeling the WS ;
– generating tests from the model ;
– executing the tests ;
– using results of those tests for certification.

Now, we present each of those steps.

2.1 Modeling the web service

Related modeling solutions

Several tools have been used in order to model WSs (with or without tests purpose). There exists standard languages in WSs : WSDL which describes the WS, and BPEL which allows to model compositions made by a WS.
Those languages are specialised and this is a limitation. To model the whole WS behaviour, we need at least three models : one written with WSDL for its description, one using BPEL describing its communications with other WSs and a third one showing its temporal evolution.

For this third model, a solution is to use FSM[1] like in [**?**]. This article presents an enhancement of WSDL, in which evolution of the WS is modeled using FSM. This article also proposes a procedure to transform WSDL into EFSM (extended FSM). Another system of derivation has also been proposed in [8].
FSM and WSDL is not a complete solution, as it does not handle compositions (unless a third model is added, such as BPEL). UML based solutions could be able to handle those three aspects of a WS.

Merging UML and WSs have already been subjects to researches. In [**?**], relations between UML meta models and WSs dedicated languages is done. No guide for models is introduced, contrary to the one which can be found in [**?**]. The proposed solution is not relevant for our test tool for two reasons : behaviour of WSs is not modeled (neither temporal evolution nor compositions) and the test tool can not understand used models.
UML model for testing WSs are also introduced in [**?**]. A component diagram is used to model the different parts of the WS, and STS[2] describes the behaviour of the WS. UML has also been proposed in [5], in order to model web sites (and not WSs). This solution cannot be directly applied to WSs, but some sub parts may be useful, such as sequence diagrams which model interactions between users.
Another UML modeling solution is proposed in [**?**]. The goal is not to produce tests, but to generate OWL-S files which models the behaviour of a WS.

[1]Finite State Machines
[2]Symbolic Transition System

We present how to create this kind of model for WSs. This is done in four steps, each one modeling a part of the WS : data it uses, behaviour of the WS, its temporal evolution and its initial state. The first step is obvious. It is creation of a class diagram which is used to model data used by the WS. The second step concern behaviours.

Modeling web services operations behaviour
The class diagram is also used to represent WSs. Each WS (the WS under test and those which may be composed) is modeled as a class, and each operation is modeled as a method of this class.
The class name is based on the WS URL (www.example.com/onlineStore/ for example). To know which WS is the subject of the test, the class name is prefixed with "sut : :".

When classes are set up, the vendor has to create the OCL code which models operations behaviour. There exists two paradigms to create OCL code. The first one uses pre-condition to forbid invocation of the operation if the system is not in the right state (defensive modeling). The second one transforms pre condition into "if" statements in post conditions, in order to raise error if invocation has not been done at the right time (offensive modeling).
Table 1 shows OCL code for the **publish** operation of our example.

LTD uses post conditions to compute test goal. In the previous example, only one test goal will be produced if we use defensive modeling (the test will check if the article is added to the cart). With the second modeling solution, three test goals will be produced : two which activate errors behaviour, and one for the success case.
Offensive modeling is more adapted for test, as all behaviour of operations are checked.

At this time of modeling, we know how each operations works, but we can not know how the service evolves. We explain now how to model this evolution.

TAB. 1 – Defensive and offensive modeling

Defensive modeling	
Pre	**Post**
self.post.state = postState::prePublished	self.post.state = postState::published
Offensive modeling	
Pre	**Post**
	if (self.post.state = postState::prePublished) **then** self.post.state = postState::published b**result** = publishErr::ok **else** **result** = publishErr::badState **endif**

Modeling temporal evolution
As the first step of modeling, modeling temporal evolution of a WS is not mandatory for all WSs. This step is only needed for stateful[3] WSs.

[3]A stateless WS has no temporal evolution. Behviour of a stateful WS will change depending on

To model the temporal evolution of a WS, we use a state-chart diagram. It has to represent every sequences of operations that can be done by a customer, not only a nominal case.

Figure 2 represents the full state-chart and a nominal one for our example (for readability reasons, only name of operation called are written, not the parameters). State chart must be complete for two reasons : produced tests cover all behaviours of the WS under test, and all temporal dependencies will be found.
At this time, the model is almost finished. The last step is to instantiate data modeled in the first phase of modeling.

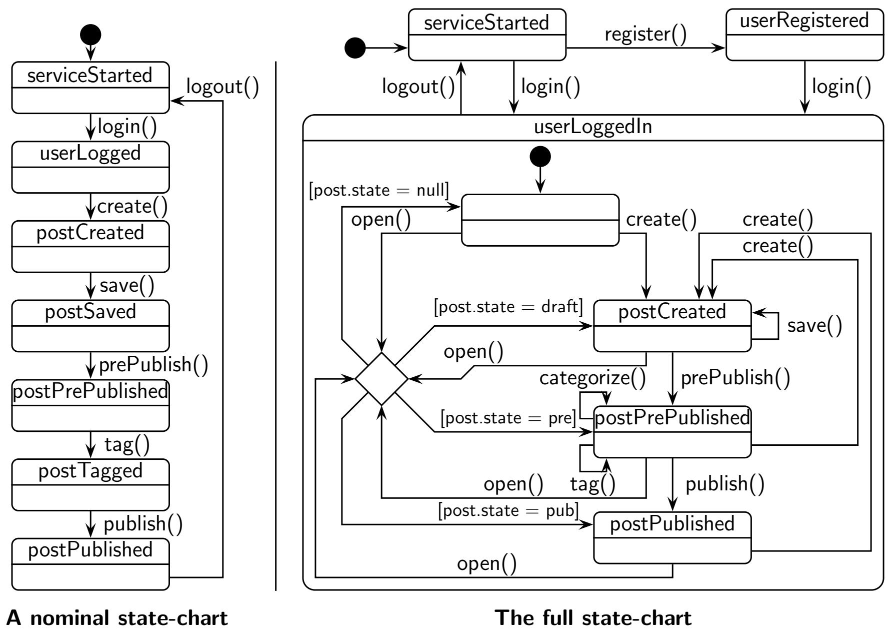

Fig. 2 – Comparison between a nominal and the full state chart

Modeling initial state
Initial state of the system is an instance diagram. In its minimal version, it must at least contain one instance of the class which models the WS.
In the first step of modeling, classes have been created to represent data used, but no real value have been stored. The goal of the initial state is not only to represent every data, but to facilitate test generation.
There is no need to represent all data, that would be useless. The most important thing is to have representative data.

Once produced, this model is used to generate tests.

2.2 Generating and executing tests

We decide to validate WSs with a model based approach. The main reasons are :

- validation must be done in the normal conditions of use (the same that a customer have when he uses a WS) ;

customer's actions

- team experience in the following area :
  - model based testing [2] ;
  - reification process ;
  - distributed software.

We use LTD (Leirios Test Designer [4]) with this model to produce the tests. The first stage is the computation of behaviour of the model. A behaviour is :

- a transition of the statechart diagram ;
- a case described in OCL clauses.

We associate a test target to each behaviour. The second stage consists in computing a sequence of method's call to put the system from the initial state to a state which activates the test target.
The last stage is to translate (reification process) the abstract test sequence into a concrete call on the WS. So, we compare the result given by model (oracle) with the result given by WS to decide if the test is correct (verdict). The comparison is fully automatic because verdict and oracle are translated during reification process.

An important question about test execution is "Who will execute the tests ?". To answer this question, we see three solutions :

- the WS vendor executes tests himself : the advantage of this solution is that tests have to be run only once. But it gives no guarantee for customers, the vendor can lie on results of the tests ;
- the customer runs the tests before sending its datas : this solution is greedy in computing power and bandwidth, constraining WSs to execute redundant tasks, producing unused results. Another problem is that WSs may not be free, and tests will cost to customer. The advantage of this solution is that customer is sure of the WSs quality ;
- a third entity takes in charge the validation : this solution is thrifty in bandwidth and computing power, as tests are only executed once. But the customer confidence problem is just shifted. In this solution, the testing entity has to prove its neutrality toward WSs vendors.

As shown before, model based testing solutions for WSs have already been proposed in [**?**] or [7], but it is not the only way to test WSs. The solution introduced in [10] presents a test system based on pool of similar WSs. All operations are called with the same parameters, and the most common answer is considered as the right one. We did not use this solution for two reasons : the first one that we may not be able to find multiple similar operations, and the fact that most operations produces the same results does not prove that this result is the right one.

So, we present our solution to give informations on WSs for futur customers.

2.3 Certifying quality

Two solutions are known for ensuring quality. The first one is introduced in [3]. In this paper, only WSs which are compliant with their requirements are published on UDDI server. This solution ensures customers that results produced by WSs they found on the UDDI server are safe, but it reduces the offer of WSs.
A second solution would be to have an evaluation of each WSs, and give it to customer. According to our knowledge, this solution has not yet been set up for

[4]www.leirios.com

WSs, but exists in the component area. In [1], about forty criteria are used to evaluate components. A survey of quality insurance for component is done in [4]. A certification based on test is proposed, in which certificaton is done when component is free of bugs.

We have chosen the second solution, because it permits to customer to choose the level of quality they want, and it does not impoverish the offer of WSs.
We exhibit now our solution for marking WSs.

The first step is to give each WS's operation it own mark. We first thought to a system based on four marks : **0** if the operation has not been tested yet, **1** if all tests failed, **2** if only some tested failed and **3** if all tests are succesfull. This solution was simple and permitted to customers to quikly identify quality of WSs. But the problem is that the mark **2** is not really representative. If we take for example two operations a and b doing the same work. For each operation, ten tests are generated and executed : one test fails for a, and seven fails for b. The two operations will obtain the mark **2**, even if b is less efficient than a.
To solve this problem, we decided to use percentage of successfull tests as mark. This makes marks more representative of tests results, and customers can more easily choose between two concurrent operations. To represent the **0** mark, a negative mark is given to the operation, in order to dissociate this case to the case in which all tests have failed.

This approach is theoritical, and needs an implementation to be validated. We introduce now our validation framework based on this solution.

3 Our validation framework

Our validation framework is based upon an UDDI server. UDDI based solution has first been introduced in [11]. This idea was also used in [3], in which a UDDI server tests WSs before their publication. It tests WS behaviour on client side, but also uses proxies to know how the WS under test acts toward WSs in case of composition. Another testing UDDI server is introduced in [12]. Contrary to our solution, it is not a model of the WS which is used to test it, but a set of scenarios describing use cases.
For our solution, we chose to rely on a UDDI server because it is used by both customers (who seek for WSs) and WSs (which wants to be known). The second reason is that UDDI server is neutral towards customers and WSs, and can stand as the third entity described before for tests execution.
Figure 3 presents an overview of our framework. We present now some steps of the framework life cycle. Phases 1 and 3 are not discussed, as they are not different that with a classical UDDI server (except declaration, but the only difference is that the UML model is given with the WSDL file). Phases 2 and 4.3 have already been presented.
We introduce now the step 4.1 : dependencies discovery.

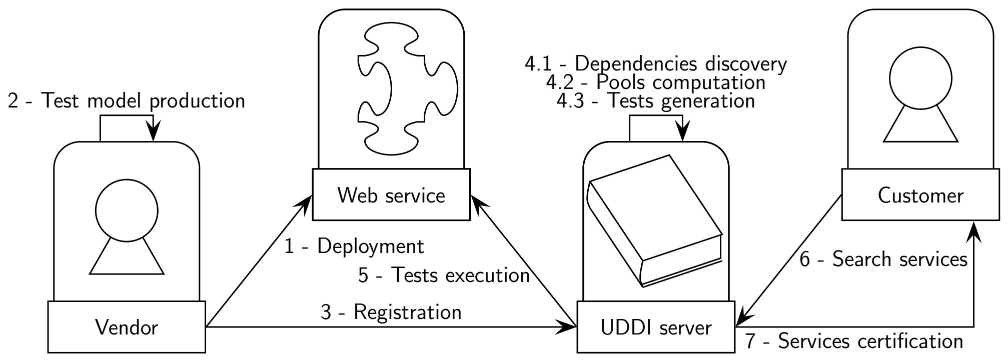

FIG. 3 – From web service deployment to customer certification

3.1 Discovering dependencies

Dependencies are extracted from the UML model given by the WS's vendor. First, compositions are seeked.

Extracting compositions

Compositions are found in operation post conditions. To discover them, we use a regular expression on OCL code which finds operation calls. Those calls represents synchronous compositions, in which results produced by composed operation are used.

Only those compositions impact on results' quality, as they can be responsible of error propagation. We do not need to change modeling to handle other compositions.

Now, compositions are found and our tool seeks for temporal dependencies.

Finding temporal dependencies

Temporal dependencies can not be automatically found in the UML model. We first thought that the state-chart could be used to find "each time" temporal dependencies : this is true in some cases (when a sequence of operations is written in the state-chart, and that there is no way to get into the sequence), but not for all. If we take a look at the state-chart of our example (figure 2), we can see that if there were not the **open** operation, the sequence **create** - **prePublish** - **publish** could be automatically found, and thus temporal dependencies between those operations discovered.

To solve this problem, our proposal is to change the model. Temporal dependencies will be described by a set of sequence diagrams, each of them specifying a temporal dependency. Figure 4 depicts the temporal dependency between operations**login** and **register**.

All these diagrams must belong to a package named "temporalDependencies". This avoids our tool to explore sequences diagrams that role is not to describe temporal dependencies.

Table 2 show dependencies found using our method. Using those dependencies, we can compute, for each operation, the pool of operations on which it depends.

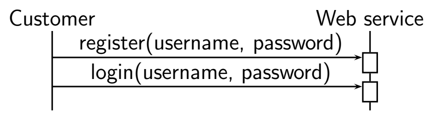

FIG. 4 – Temporal dependency declaration

3.2 Creating pools of operations

For each operation, we define a pool, which is composed of all operations on which it depends : for each operation x, we define the pool P_x. We consider S as the set of all known operations, and x a member of it.
The algorithm we propose to compute P_x can be splitted in two steps. The first one is to define the R dependency graph : vertices are operations, and each edge represents a relation between two operations. These edges are oriented, according to the relation's direction (for example, if a composes b , then there will be an edge starting from the vertice representing the operation a to the vertice representing the operation b). Figure 5 represents the graph made from our example. The second step is to compute the transitive enclosure of R to produce the R' graph. For each operation x, we define P_x as the set of all neighbours of x in R'.

Using this solution, the pool of an operation x contains all operations on which it depends, directly or not.

Now, we apply our pool reckoning on our example. Table 2 presents relations linking operations and pools computed using these relations. Knowing this, we can start validation for each operation.

TAB. 2 – Relations between operations and pools of dependencies (* indicates an "each time" temporal dependency)

Operation	Composes	Temporally depends on	Pool
Blog framework			
register	-	-	-
login	-	register	register
logout	-	login	login, register
create	-	login	login, register
save	-	create	create, login, register
open	-	save	save, create, login, register
prePublish	-	create	create, login, register
tag	-	prePublish*	prePublish, create login, register
categorize	-	prePublish*	prePublish, create login, register
publish	ping	prePublish*	ping, prePublish, create login, register
Blog directory			
ping	-	-	-

These pools are used to compute marks for each operation. To compute those marks, we first need to test operations.

3.3 Tests execution

As explained before, tests are generated from the UML model. These tests are given to testing agents, which are distributed through the Internet. We have chosen agents because of their communication habilities and their capability to travel on the network. These capabilities are useful to prevent overloading of a single computer executing tests.

Order of test is defined from relations linking operations. First, we isolate operations that have no dependencies. They will be tested first.
The second test wave is composed of operations which only depend on already tested operations. We repeat this until all operations have been tested. Dividing operations into test waves can be done because there can not be cycles in the dependency graph (their existence would lead to infinish loops - a calls b, which calls a ...- or impossible calls - a can not be called if b has not been called yet, which can not be used if a has not be used before ...).

Figure 5 represents the order of test in our example. Edges are the relations between operations, and each layer corresponds to a testing wave (the lower layer is the first test wave, the layer upon is the second wave and so on). Inside a testing wave, order of test has no importance.

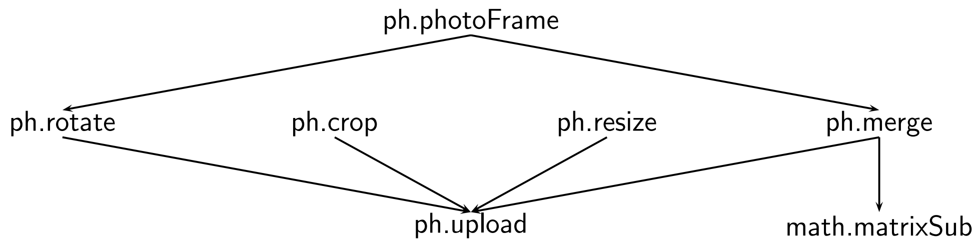

FIG. 5 – Graph of dependencies between operations, sorted to represent testing waves

Once all tests are executed, each agent sends results to the coordinator agent. It uses these results and compare them to oracle computed from the model. Results of these comparisons are used to give a temporary mark for each operation. In order to have the real mark, the coordinator agent computes pool for each service. For each pool, the mark given is equal to the lowest mark of all operations included in the pool. In this way, if an operation is unreliable, its mark influences the mark of the pool, even if it does not directly influence other operations depending of it. This makes customers able to see that there is a problem. Once all pools and marks computed, they are used to update marks given to operations.
For our example, results of tests and obtained marks by operations are described in table 3.

TAB. 3 – Marks obtained by operations and pools

Operation	Single mark	Pool mark
Online photo tool		
ph.upload	100	100
ph.resize	100	100
ph.crop	73	73
ph.rotate	40	40
ph.merge	90	90
ph.photoFrame	45	40
Mathematical WS		
math.martixSub	100	100

Now, we present the way the framework interacts with customers.

3.4 Finding and using WSs

When a customer searches for an operation to use, he gives its description to our UDDI server, like in a normal customer-UDDI conversation. But he also gives his requirements :

– wM, the mark wanted for operations;
– mM, the minimal mark for operations it there exists no operation corresponding the wM.

When our UDDI server receives the request, it find in the database all operations corresponding to the description. Then it filters this list to obtain only operations having a mark above wM. If this filters returns an empty list, a new filter is ran on to find operations aving a mark above mW.
The produced list is sent to the customer, who picks the most relevant operation. Then he acts with the WS in a classic way.

4 Conclusion and future works

In this paper, we have presented a solution to solve confidence problem between WSs and customers. We first introduced why customers can not trust WSs. Then, we proposed a theoritical approach to solve this problem, and at least a framework implementing this method.
The theoritical approach relies on model based testing. First, we have proposed an UML modeling solution which presents two major interests. First, it uses UML which is a wide spread language. This may facilitate the use of our method by industrial actors. The second advantage is that our model not only describes the functional behaviour of the WS, but also dependencies existing between services. Thus, it is not mandatory to produce three models (one for behaviour, another for composition and a last one for temporal dependencies). This implies a time gain during modeling, but the produced model can not be as expressive as dedicated one. A solution could be to introduce extensions to UML, but models would not be as simple to create as nowadays.

This model is then used to produce tests. Result of those tests are then used to produce a mark for each operations : the mark represents the successful tests ratio. This permits to customer to easily evaluate quality of an operation.

The theoritical approach has been then brought to a real validation framework, based on an UDDI server. When a WS declares to it, he joins an UML model to its WSDL description. This UML model is used to discover dependencies between WS operations : those dependencies are then used to create pools of dependent operations and to know order in which tests have to be executed. Then, we use the UML model with a testing tool (LTD), to produce tests which are executed by testing agents dispatched throughout Internet. They send results of the tests to a coordinating agent which computes the mark for each operation.
When a customers asks to our UDDI server for WSs, it sends the list of corresponding operations and their marks. This makes the customer aware of the quality of the operation he is going to use.

At this time, the framework is not fully functionnal. We already have a tool for dependencies discovery from UML model, which has been tested on real industrial cases.
We are currently working on a second tool which aims to merge different WSs models. This tool will be used when we have to test a WS which operates distributed compositions (operations involved in the composition do not belong to the same WS). Once this tool produced, we will focus on a automatic reification system. It will be able to transform abstract tests produced by LTD into test agents, wich will execute the tests.

Références

[1] Alexandre Alvaro, Eduardo Santana de Almeida, and Silvio Romero de Lemos Meira. Quality attributes for a component quality model. In *Tenth International Workshop on Component-Oriented Programming*, Glasgow, Scotland, 2005.

[2] E. Bernard, F. Bouquet, A. Charbonnier, B. Legeard, F. Peureux, M. Utting, and E. Torreborre. Model-based testing from UML models. In *MBT'2006, Model-based Testing Workshop, INFORMATIK'06*, volume P-94 of *LNI, Lecture Notes in Informatics*, pages 223–230, Dresden, Germany, October 2006. ISBN 978-3-88579-188-1.

[3] Antonia Bertolino and Andrea Polini. The audition framework for testing web services interoperability. In *EUROMICRO-SEAA*, pages 134–142. IEEE Computer Society, 2005.

[4] Xia Cai, M. R. Lyu, Kam-Fai Wong, and Roy Ko. Component-based software engineering : technologies, development frameworks, and quality assurance schemes. In *APSEC '00 : Proceedings of the Seventh Asia-Pacific Software Engineering Conference*, page 372, Washington, DC, USA, 2000. IEEE Computer Society.

[5] Ana R. Cavalli, Stéphane Maag, Sofia Papagiannaki, Georgios Verigakis, and Fatiha Zaïdi. A testing methodology for an open software e-learning platform. In *EDUTECH*, pages 165–174, 2004.

[6] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Wsdl 1.1. http ://www.w3.org/TR/wsdl, 2001.

[7] Philip Mayer and Daniel Lubke. Towards a bpel unit testing framework. In *TAV-WEB '06 : Proceedings of the 2006 workshop on Testing, analysis, and verification of web services and applications*, pages 33–42, New York, NY, USA, 2006. ACM Press.

[8] Avik Sinha and Amit Paradkar. Model-based functional conformance testing of web services operating on persistent data. In *TAV-WEB '06 : Proceedings of the 2006 workshop on Testing, analysis, and verification of web services and applications*, pages 17–22, New York, NY, USA, 2006. ACM Press.

[9] OASIS UDDI specification TC. Uddi version 3.0.2. http ://uddi.org/pubs/uddi_v3.htm, 2005.

[10] Wei-Tek Tsai, Yinong Chen, Raymond A. Paul, Hai Huang, Xinyu Zhou, and Xiao Wei. Adaptive testing, oracle generation, and test case ranking for web services. In *COMPSAC (1)*, pages 101–106, 2005.

[11] Wei-Tek Tsai, Raymond A. Paul, Zhibin Cao, Lian Yu, Akihiro Saimi, and Bingnan Xiao. Verification of web services using an enhanced uddi server. In *WORDS*, pages 131–138. IEEE Computer Society, 2003.

[12] W.T. Zhang Tsai, Y. D., Chen, H. Huang, R. Paul, and N. Liao. Scenario-based web service testing with distributed agents. In *IEICE Transaction on Information and System*, pages 2130–2144, 2003.

[13] Werner Vogels. Web services are not distributed objects. *IEEE Internet Computing*, 7(6) :59–66, 2003.