Automated UML models merging for web services testing

Vincent Pretre, Adrien de Kermadec, Fabrice Bouquet, Christophe Lang, Frédéric Dadeau

Automating UML Models Merge for Web Services Testing

Vincent Pretre*, Adrien de Kermadec, Fabrice Bouquet, Christophe Lang and Frédéric Dadeau

Laboratoire d'Informatique de Franche-Comté
16 route de Gray, 25030 Besancon CEDEX, FRANCE
Fax: +33 381666450
E-mail: vincent.pretre@lifc.univ-fcomte.fr
E-mail: adrien.dekermadec@lifc.univ-fcomte.fr
E-mail: fabrice.bouquet@lifc.univ-fcomte.fr
E-mail: christophe.lang@lifc.univ-fcomte.fr
E-mail: frederic.dadeau@lifc.univ-fcomte.fr
*Corresponding author

Abstract: This paper presents a method for merging UML models which takes place in a quality evaluation framework for Web Services (WS). This framework, called iTac-QoS, is an extended UDDI server (a yellow pages system dedicated to WS), using model based testing to assess quality. WS vendors have to create UML model of their product and our framework extracts tests from it. Depending on the results of the test execution, a mark is given to WS. This mark gives to the customers an idea about the quality of WS they find on our UDDI server. Up today, our framework was limited to WS which did not use other WS. This was justified by the fact that it is impossible for vendors to create a good model of a foreign product. Our method for model merging solves this problem: each vendor produces models of its own product, and we automatically merge the different models. The resulting model from this merging represents the composition of the different WS.
For each type of diagram present in the models (class, instance or state-chart diagram), a method is proposed in order to produce a unique model. In addition to this, a solution is proposed to merge all OCL code in the class modeling the WS under test. Unfortunately, this process introduces inconsistencies in the resulting model, that falsify the results of the subsequent test generation phase. We thus propose to detect such inconsistencies in order to distinguish inconsistent and un-reachable test targets.

Keywords: Web services, UML, model based testing, consistency

Reference to this paper should be made as follows: Pretre, V., de Kermadec, A., Bouquet, F., Lang, C. and Dadeau, F. (2009) 'Automating UML Models Merge for Web Services Testing', *Int. J. of Web and Grid Services*, Vol. 5, No. 1, pp.xxx–xxx.

Biographical notes: Vincent Pretre received his PhD (2009) from Franche-Comté University, France. His research works are mainly in the domain of web services quality and especially test generation and model merging.

Adrien de Kermadec is a PhD candidate in the Department of Computer Sciences at Franche-Comté University, France. Adrien is a member of INRIA Cassis project. His research interests include model quality like model-based testing and model checking using constraint logic and prover.

Fabrice Bouquet is professor at the University of Franche-Comté and member of INRIA Cassis project. He received his PhD (1999) from the University of Provence, France. His research interests are in the field of modeling quality and usages in particular the test generation.

Christophe Lang is an assistant professor at the university of Franche-Comté. His research works are mainly in the domain of distributed systems and especially multi-agent systems and web services.

Dr. Frederic Dadeau is assistant professor at the University of Franche-Comté. His work mainly focus on model-based test generation using constraint logic programming and solving.

1   Evaluation of web services

Since few years, Service Oriented Architecture takes an important place in software development. Web Services (WS) are used in many kinds of applications: web sites, widgets or more classical software. This omnipresence leads to a need for validation, that led us to create a quality evaluation framework for WS. Until now, this framework was not able to test WS working with other WS. This paper introduces a method for model merging, which addresses this issue.

The first section introduces context which led us to propose a model merging solution. First, we introduce WS and our vision of their quality. Then, we present iTac-QoS, our validation framework for WS. The merging method proposed in this paper takes place in this framework. At last, we present why model merging is a necessity for testing a particular type of web services.
Second section is dedicated to related works. It is split in three parts: UML modeling of WS, test of WS and model merging. The third section introduces our method to automatically merge UML models, applied on an example.
Then, the fourth section presents how to detect and solve the inconsistency problems resulting from the model merging process.
At last, we conclude and introduce future work.

Figure 1   From Web Service deployment to quality assessment

1.1   Web Services Quality

We define WS as the server part of a client-server application in which exchanged messages are written in XML (Vogels, 2003). Generally, WS are built upon SOAP (Simple Object Access Protocol (Gudgin et al., 2007)). This protocol defines structure of XML messages, and uses well-known protocols such as `http` or `smtp`.
The first advantage of WS is their compliance with most security policies, which generally authorize http and smtp protocols. Moreover, these protocols and XML bring the second main advantage of WS: portability. Almost every programming language has libraries handling XML and those protocols. This makes it possible to easily create clients for a web service in any programming language.
Writing a client for a WS can be easily done using a WSDL (Web Service Description Language (Christensen et al., 2001)) file. This file describes everything required to communicate with a WS: address, provided operations, message format, exchange protocols etc. A second technology related to WSDL is UDDI (Universal Description Discovery and Integration (OASIS UDDI specification TC, 2005)). A UDDI server is a yellow pages system for WS. It permits customers to easily find WS, and vendors to register and diffuse their products. We are currently working on an improvement of UDDI, in which the server provides more than the information given in the WSDL files, it also assesses the quality of results produced by WS.

There exists many criteria to describe quality of a WS: accessibility, time needed to compute a result, privacy of data and customer, quality of results etc. We consider two criteria to define the quality of a WS. The first one is results correctness (is the answer given fits with the expected one?) and the second one is result completeness (if many answers can be given, are they all provided?).
There is many reasons that can lead to quality leak of a WS. We can classify them in two categories: static or dynamic, depending on the reproducibility of the quality leak. Reproducible problems are considered as static, the others as dynamic.
For WS, the most obvious dynamic problem is related to the network: packets may be lost during transport, leading to a corrupted message. As this kind of problem

is not reproducible, it is quite impossible to certify that a WS is free of them.
There exists few static reasons for quality leak. The first one is, obviously, bugs
introduced during the development phase. The second one is software on which the
WS relies: web server, libraries, other applications, etc. Bugs may exist in those
software, and have an impact on WS quality. The last reason is the interaction
relations between WS operations.

There exists two kinds of relations between WS operations. The first kind of re-
lation is **composition**. An informal definition of composition is "a WS operation
acting as a client of another operation". There exists several compositions. We
consider three criteria to discriminate them: **synchronicity** (a composition is syn-
chronous when the composing operation uses results of the composed operation),
distributivity (a composition is distributed when the two involved operations do
not belong to the same WS) and **dynamicity** (a composition is dynamic when the
composed operation has been found through a UDDI server). This last criterion
depends on distributivity of a composition (a composition can be dynamic only
if it is distributed). Due to fault propagation, synchronous compositions have an
impact on quality. If an operation bases its results on wrong data, it may be false.
The second kind of relation is the **temporal dependency**. Its informal definition
is "operation a temporally depends on the operation b if it can be called only if b
has been called previously". Generally, this dependency is due to shared data, on
the server side (operation b writes in a database, operation a reads what has been
written) or on the client side (operation b produces a list of results, the client picks
up one and uses it as a parameter of a). As for composition, temporal dependency
can be responsible of fault propagation.
Thus, as we see in this part, there are numerous possible quality leaks in WS. This
is why we set up iTac-QoS, a framework which validates functionnal quality of WS.

1.2   iTac-QoS: using tests to asses Quality

iTac-QoS (iTac Tests And Certifies Quality of Services (Pretre, Bouquet and
Lang, 2007)) is a quality evaluation framework for WS. It relies on a UDDI server,
and provides quality information based on tests. Figure 1 shows the evaluation
process, that we now describe.
First, the WS vendor deploys its product (1), and creates a model for test (2).
When he registers its product on our framework (3), provides classical information
(WSDL file, description etc.) and the model. Before test production, the frame-
work performs an analysis of the model (4.1) such as interaction discovery, or string
replacement. If distributed compositions have been discovered, models of the dif-
ferent WS need to be merged (4.2). Then, tests are generated (4.3) using Test
Designer (TD) a commercial tool from the Smartesting company[a]. and executed
(5). Results of these tests are used to compute a mark for each WS operation.
When a customer searches WS on the UDDI server (6), this latter provides the list
of corresponding WS and the mark they obtained (7). This mark helps customers
to have an idea of the quality of WS found.

---

[a]www.smartesting.com

This paper focuses on step 4.2. In order to make our method for model merging easier to understand, we will detail two important steps of the assessment process: modeling and test generation.

The modeling solution chosen was UML, because of its extensibility and reputation. WS vendors are more likely to know UML than any other modeling language (Eid, Alamri and El Saddik, 2003).

UML models required by our framework contain three types of diagrams (**class diagram**, **instance diagram** and **state-chart diagram**), completed by **OCL code**. Those models have to be compliant for the test generation tool we consider, Test Designer (TD), as described in (Bouquet et al., 2007b).

The first type of diagram is class diagram. Class diagram is mandatory for two different parts of the WS: interface and data used. The interface part is composed of a class representing the WS and a set of class describing exchanged messages. This part of the class diagram is needed to produce suitable test scripts (in which messages content is set before operation call). The second part of diagram class is more classical, and represents data used by the WS (for example, a model describing an online store WS will contain classes "customer", "commands", "products" etc.).

For each operation described in the class diagram, post-conditions must be expressed using OCL. They describe how the model changes when the operation has been called. We do not use pre-condition (meaning that the pre-condition is always true). Our model is built upon the defensive paradigm: error cases have to be handled in post-conditions and assign error codes to specific variables. We chose this solution in order to be able to capture all the possible behaviors of the WS under test.

The second diagram of our model is an instance diagram. When the class diagram gives structure of data, instance diagram gives value to them. This diagram has to present real data, because our framework is fully automatic, and nobody will translate abstract data into real data. All classes declared in the model must have at least one instance in the instance diagram. It is useless to represent all data used by the WS in the instance diagram. The modeler has to choose a subset of data and integrate them in the model. Data chosen must be as representative as possible. For example, if the modeled WS handles multiple level of users, the model must contain at least one user of each level (such as admin, member, guest, etc.).

The last kind of diagram is state-chart diagram. Its goal is to model the life-cycle of the web service. This is not a mandatory diagram, but it helps test generation by describing how WS has to be used. If such a diagram is made, it must represent all possibilities of WS utilization.

Figure 2 synthesizes how a WS is decomposed in its model.

We then use Test Designer (TD) to generate tests. First, it extracts every behavior described in the model. A **behavior** can be seen as a path in the control-flow graph of an operation written in OCL (for example, an "if" statement may create two behaviors: one when its condition is satisfied and one when it is not). Behaviors may also be extracted from a state-chart diagram (when a state is reached).

All those behaviors are called **test targets**. For each one, TD generates a sequence of operations that aims at activating it. This sequence is called a **test**.

An **oracle** is associated to each test. It is given by the state of the model after test

execution and/or the values returned by the operation calls. In our framework, we only consider the latter option: if the results produced by the last operation of the test are not the expected ones, the test fails.

Now, we present the limitations of this modeling solution and why it is not adapted for WS composition testing.

1.3   *The Necessity of Model Merging*

The need for model merging only occurs when a WS uses other WS. To test such a WS, there are two solutions. In the first one, the vendor of the WS produces a model describing the two WS involved in the composition. The second solution is that each vendor models its own product, and our framework merges these two models. The result of this merging is a new model which described the composition of the two WS.
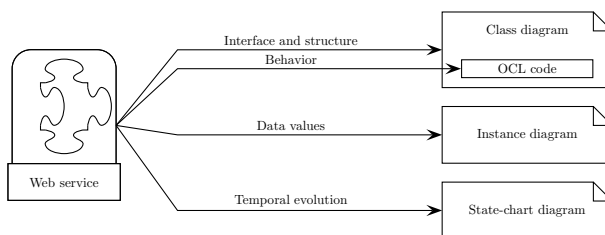
We consider that the first approach is not realistic, since a vendor may not be able to produce a valid model of a foreign product. A "login" operation seems simple to model: if there exists a user corresponding to login and password given in parameter, he is logged in the system. If there is no such user, an error is raised. This solution would certainly fit in most cases. But if the WS has stronger security policies, tests may fail because the model does not match with implementation. For example, composed WS security policies specify that a customer can not be logged in twice in the same time. If our framework is not aware of this, it will try to run parallel tests to accelerate quality evaluation process, and tests will fail.

Security policies are one of the reasons why model of the composed WS must concord to its behavior. A simple customer is aware of how the composed WS reacts in normal conditions of use, but may not precisely know all security policies. For this reason, he is not able to produce a efficient model for this WS.

A second problem of modeling a foreign WS is creation of instance diagram. As we said before, this diagram must represent real data. Someone modeling a foreign WS may not know which data it uses, and so may be unable to produce a useful instance diagram. As we explained before, the instance diagram has to be representative of the system. If data is missing, the testing tool may not be able to produce some tests.

A solution could be to get model given by the vendor when he registered his

Figure 2   An overview of model's structure

WS. As we saw before, this model contains real data that may be critical. For this reason, models can not be shared. Thus, this solution can not be used.

Those two reasons led us to choose the solution of model merging. Before introducing our method, we present works about WS testing and model merging.

2   Related works

In order to propose an efficient solution for model merging, we had to look at existent solutions. This section is divided in three parts. First, we introduce works using UML to model compositions. Then, we take a look at works using UML to test WS, and how they handle compositions. Finally, we present works about merging UML models.

2.1   UML for modeling web services and compositions

UML-WSC is an UML extension introduced in (Thöne, Depke and Engels, 2002). It is presented as an alternative for BPEL[b]. Another solution for composition modeling has been introduced in (Benatallah et al., 2003). This paper presents design patterns to easily design compositions with UML.

We think that these works are not relevant for our purpose. The first reason is that those models are not designed for test production. The second one is that these models represent the whole composition. We explained before why we do not think that this kind of method is suitable for our framework.

UML modeling for WS was also presented in (Marcos, de Castro and Vela, 2003). This modeling solution is not suitable for test generation, because behavior of WS is not modeled. A link between UML meta-models and WS standards (WSDL, BPEL) is shown in (Staikopoulos and Bordbar, 2005).
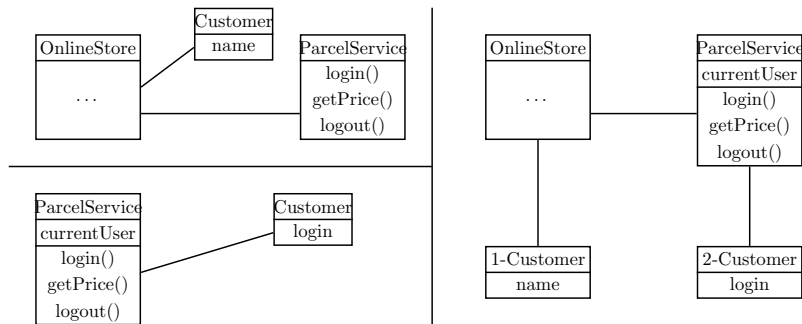
2.2   Testing web services with UML

The choice of UML for test generation can be criticized, because of its lack of formalism. For example in (Heckel and Mariani, 2005), UML has been set aside because authors considered that UML could not describe precisely the evolution of a WS. Our modeling solution does not suffer from this lack of formalism, as it has been shown in (Bouquet et al., 2007b). This paper has proven that the UML subset used in TD is non-ambiguous.

There also exists works dealing with UML modeling for WS testing. Some does not take composition into account, such as (Frantzen, Tretmans and de Vries, 2006). Others deal with composition in different ways. In (Heckel and Lohmann, 2005), when the WS under test performs a composition, it does not really communicate with another WS. The test driver emulates the composed WS. In (Bertolino and Polini, 2005), a proxy is set up between WS under test and composed WS. That allows to verify how the WS under test acts toward composed WS.

In (Foster et al., 2003), UML is combined with BPEL in order to verify if compo-

---

[b]Business Process Execution Language - A language used to model compositions

Figure 3   Example of class diagram merge

sition workflow is respected.

Those solutions are really efficient for composition testing, but do not provide clues to our main problem: how to easily make a model of the composition compliant with our test tool?

According to our knowledge, there is no solutions dealing with model merging to handle WS testing. To propose a solution, we had to investigate on works about model merging.

2.3   Model Merging

Composition is not a specificity of WS. Since a few years, different solutions have been proposed for creating applications from components. Having models to represent those compositions is one goal of (Bézivin, 2001). This paper does not show solutions for model merging, but deals with the use of meta-models to handle several kinds of components (EJB, Corba etc.).

In (Estublier and Ionita, 2005), a UML profile is proposed in order to help model compositions. No explanations are given on how to produce an unique model from different models using this profile.

A model merging solution is introduced in (Clarke, 2002). This method aims at helping subject-oriented design. An application is not designed by a big model, but by a set of small models. Each one of those models describes a subject of the application. The method proposed in this paper may not be applied to WS model merging, because composition description requires both models. In (Boronat et al., 2007), another model merging solution is proposed. This solution only applies a merging for diagram classes, and do not take into account other types of diagram. We present now our own solution for model merging.

3   Merging UML models

We saw that model merging is necessary to test a WS performing distributed compositions. In order to illustrate our method for model merging, we introduce an example.

This example is composed of two WS. The first one is an online store WS, which composes a parcel service WS. The online store WS provides operations: `register`, `login`, `logout`, `search`, `addToCart` and `validate`. The parcel service WS provides three operations: `login`, `getPrice` and `logout`.

We now present our method for model merging, which is done in four steps: the first one transforms state-charts into OCL code. The second one consists in merging the different class diagrams. Then, OCL code of classes are merged in class modeling the WS under test. Eventually, instance diagram are also merged to produce the final model.

3.1   State-chart

There would be two solutions to produce a final model in which specifications of the several state-charts are captured. The first one would be to produce a final state-chart that is a combination of the different one.
The second solution would be to translate the state-charts of composed web services into OCL code, and the state-chart of the composing web services is kept in its original version.

With the first solution, the final state-chart models temporal evolution of every WS involved in the composition. With the second one, the state-chart only models evolution of the WS under test.
This is why we chose this last solution.

The first step in the transformation of the state-chart in OCL is to create a new enumeration ("state-List"), which lists all states described in the state-chart, including initial and final state. A new attribute ("state") is added to the class describing the web service (and its type is the enumeration previously created).
Then, for each operation, we list all transitions using it as a trigger. These transitions are noted as 4-tuple, in the form {lS, g, rS, op}, where lS is state leaved by the transition, g the guard of the transition, rS the state reached by the transition and op operation that may be automatically activated when rS is reached.
This list is the base for OCL update, which permits simulation of the state-chart.

Update of OCL code is done in two steps. First, we include all the OCL code of the operation in a "if" statement. Its condition is "(((self.state = lS_1) and (g_1)) or ((self.state = lS_2) and (g_2)) or ... or ((self.state = lS_n) and (g_n)))". Thanks to this condition, the operation has effects only if it is called in a case that has been specified in the state-chart diagram.
Then we simulate the evolution in the state-chart diagram. To do this, at the end of OCL code (just before closing the "if" statement previously created), we create a list of "if" statement that make the "state" attribute's value change. They look like this following code:

```
if ((self.state = lS_1) and (g_1)) then
  self.state = rS_1 and self.op_1()
else
  if ((self.state = lS_2) and (g_2)) then
    self.state = rS_2 and self.op_2()
  else
```

```
...
    if ((self.state = lS_n) and (g_n)) then
       self.state = rS_n and self.op_n()
    endif
  ...
  endif
endif
```

There is a particular case when the reached state is a choice point. In this case, we can not write "self.state = ...". Each transition is transformed into nested "if" statements (where the guard of the transition becomes the condition of the "if" statement). The order in transformation has no importance, except if a guard of one of these transition is "[else]". In this case, the state reached by this transition is used to produce the "else" case of the last "if" statement. The code produced by this transformation is shown in code below:

```
if (g_1) then
 self.state = rS_1 and self.op_1()
else
  if (g_2) then
    self.state = rS_2 and self.op_n()
  else
  ...
      if (g_n) then
         self.state = rS_n and self.op_n()
      else
         /* if there is an ''[else]'' guard */
         self.state = rS_else and self.op_else()
      endif
  ...
  endif
endif
```

Once all operations are treated, there is one step left before state-chart diagram destruction. In some cases, operation may use the "OCLisInState(x)" keyword (that permits to know the current state of the system). We replace this code by "self.state = xS", where "xS" is the enumerate value corresponding to the state "x".

Now the state-chart diagram can be deleted, because it do not have any interest anymore. The next step in model merging concerns the class diagram.

3.2   Class diagram

Class diagram merging is a quite simple step: all classes of the models have to be merged into a single class diagram. The main problem is aliasing: if two classes have the same name, do they represent the same thing? Obviously, we can not trust classe names. Two classes named "customer" in two models of two different WS will not represent the same customers. But two classes named "getParcelPriceInputMsg" will represent the SOAP message used as input parameter for the `getparcelPrice` operation.
We have to discriminate classes modeling data shared between the two models (classes modeling SOAP messages and WS interfaces) and local classes. To do that, we use the WSDL file of each service. As explained before, those files describe messages used to communicate with a WS. So we use those files to discover which classes models shared messages. All other classes are renamed before merging. To rename classes, we give a number to each model. Then, for each class which do not represent a shared message, we prefix its name with the number given to the model it belongs to. Once renaming is done, all classes are put into a new class diagram.

It is possible that shared messages are not represented exactly the same way. In

this case, we keep all attributes and operations from the several classes modeling those data and include them in a single class.

Figure 3 shows merging process on a sub-part of our example. For space reasons, we simplified the diagram (for example, cardinality, name and role of association are removed). Up-left part represents a sub-part from the online store model, below is shown a sub-part from the parcel service model. The right side is what is obtained when merging these two sub-parts.
Once this step is finished, we can merge all OCL code into the class modeling the web service under test.

3.3   OCL

We propose a solution for OCL merging because our test tool (TD) does not handle operation call. That means that the whole OCL code must be contained in the class representing the web service under test.
This merging solution is only possible because TD understands OCL code as a sequential language, not as a parallel language. For example, if we write "self.attr = 2 and self.attr = 4", this should be evaluated as false, because "attr" should have for value 2 and 4 in the same time. For our testing tool, this means "The value 2 is assigned to attr, and then is overriden by 4".
This solution does not comply with OCL specifications (in which OCL is an constraint language, not an action language). This solution has been chosen during TD design to help model design: most people know how to use action language, and some operations could not be written with parallel interpretation of OCL.

Merging OCL code is done in two phases: the first one is the references update. In OCL code, there may be many references (to an attribute of the class, to another class etc.), that are linked to a context. For example, the operation "login" of the class "parcelService" has for postcondition "self.user = pUser". The context is the operation "login", so "self.user" refers to the attribute "user" of the class "parcelService".
In order to move all OCL code in the class representing the web service under test, we have to change all those references to adapt them to their new context. This is done by changing the "self" keyword by the path linking new context to old context. In the previous example, the code will be moved to the class "onlineStore". The classes "onlineStore" and "parcelService" are linked by the association "uses", the role on the "parcelService" class side is "parcelWS". When OCL code of the operations of the "parcelService" class is moved to the operations of "onlineStore", all references to "self" will be changed to "self.parcelWS".
We also have to update the second kind of references that are operation parameters. All references to parameters must be changed by the value assigned at the call.

Once this is done, OCL code of an operation is moved before its call. The next step is to link results produced by operation to call of the operation. There is two possibilities here. The first one occurs when "result" is only affected once. In this case, this affectation is deleted, and the call to the operation is replaced by the value used to instantiate "result".

In the second case, "result" has multiple affectation. Here, we create a new attribute
in the class under test, having the same type than the result. Each occurrence of
result will be replaced by this attribute, and the call to the operation will also be
replaced by this attribute.

Code below shows evolution of code in our example. We chose composition of
operation `login(user, passwd)` from the parcel WS by operation `validate` from
online store WS.

In order to make the example more readable, we had to simplify the code (we do
not use classes defining SOAP messages).

```
/*-- ORIGINAL CODE --*/
/* Operation: validate */
if (self.parcelService.login("login", "password" = "ok")
then
    /* business code, useless for the example */
endif
/* Operation: login(user, passwd) */
if (self.users->exists(u|u.login = user and
    u.password = passwd) then
  self.currentUser =
      self.users->any(u|u.login = user and
      u.password = passwd) and
  result = "ok"
else
  result = "Authentication failed"
endif
/*-- 1 -- REFERENCES UPDATE --*/
/* Operation: login(user, passwd)
    Every reference to self is changed to
    self.parcelService */
if (self.parcelService.users->exists(u|u.login = user
                            and u.password = passwd) then
  self.parcelService.currentUser =
      self.parcelService.users->any(u|u.login = user
                            and u.password = passwd) and
  result = "ok"
else
  result = "Authentication failed"
endif
/*-- 2 -- PARAMETERS UPDATE --*/
/* Operation: login(user, passwd)
    References to parameter user are changed by
    its value "login", and those to parameter
    passwd to "password" */
if (self.parcelService.users->exists(u|u.login = "login"
                            and u.password = "password") then
  self.parcelService.currentUser =
      self.parcelService.users->any(u|u.login = "login"
                            and u.password = "password") and
  result = "ok"
else
  result = "Authentication failed"
endif
/*-- 3 -- CODE MERGING --*/
/* Operation: validate()
    All code from operation login() is brought before
    its call in operation validate() */
if (self.parcelService.users->exists(u|u.login = "login"
                            and u.password = "password") then
  self.parcelService.currentUser =
      self.parcelService.users->any(u|u.login = "login"
                            and u.password = "password") and
  result = "ok"
else
  result = "Authentication failed"
endif and
if (self.parcelService.login("login", "password" = "ok")
then
    /* business code, useless for the example */
endif
/*-- 4 -- MATCHING RESULT AND CALL --*/
/* Operation: validate()
    A new attribute, called "tmpLogin", is created in the
    class modeling the online store WS.
    References to result coming from login() code are
    replaced by this attribute. The same thing is done for
    the call to the operation.*/
if (self.parcelService.users->exists(u|u.login = "login"
                            and u.password = "password") then
  self.parcelService.currentUser =
      self.parcelService.users->any(u|u.login = "login"
                            and u.password = "password") and
  self.tmpLogin = "ok"
else
```

```
    self.tmpLogin = "Authentication failed"
endif and
if (self.tmpLogin = "ok") then
  /* business code, useless for the example */
endif
```

3.4 Instance diagram

Initial state merging is done by bringing all instances of class provided in the different models into a single instance diagram. As in class diagram merging, we have to change names in order to avoid conflicts naming between. The same principles than in class renaming is applied, all names are prefixed by the number previously given to the model. As classes names have been changed, we have to update instances to fit those changes. Then, we can move all instances to a single instance diagram.

Now, we have to correct impacts of class diagram update on instances. The first action is to locate classes that received new attributes during class diagram merging. For all instances of those classes, we have to instantiate every attribute to its default value (as specified in models or arbitrarily chosen by us).
The next issue is to have only one instance of each web service interface. If we keep more than one instance of a web service interface, the model will not be realistic, because there exists only one web service corresponding the instantiated class. To do this, we only keep the instance given in the model describing the concerned web service. If we have two different instances of the class modeling the interface, we consider that the vendor of the web service knows his product better than everyone else. That means that his instantiation of the class can be more trusted, so it is the one that is kept.
The third problem may be caused by associations between classes, and particularly associations having a cardinality of "1" or "0..1". If we take our example of online store and parcel service again, the "parcelService" class has a relation with the "newParcelResponse" class, the cardinality on the "newParcelResponse" side is "1". In the two models, we have a instance of each of these class, and a instance of the relation liking them. We saw before that we only keep the instance provided by the parcel web service vendor. If we create a link between the "parcelService" instance and the two instances of the "newParcelResponse" class, our model will not be correct, because the cardinality of the association is broken. For the same reasons than before, we only keep the instance provided by the vendor of the parcel service, and the other one is deleted. If other links were existing between the deleted instance and other instances, they are reported to the instance kept.

We have to point at the fact that, in normal conditions of modeling, only interfaces of web services and messages received and produces can be found in two different models. This explains why we mainly talk about these kind of instances.

Once the new instance diagram is created, our model is ready for test generation.

3.5   Tests generated from merged model

TD is used to generate tests. This tool is based on the extraction of every behavior of the operations described in the model. It is important to know that TD has a particular way of interpreting OCL. OCL is usually a parallel language, used to describe the states of the model before and after each operation. TD considers OCL as a sequential action language, having the same syntax, and in which each equality is interpreted as an assignment modifying the state variables of the models. If this particularity is helpful for model interpretation and merging, we lose the commutativity and associativity of predicates.

The operation code has to be put in the SSA form (Static Single Assignment (Myers and Sandler, 2004)), a predicative representation to keep intermediate calculus for sequential semantics. It is obtained by adding intermediate state system variables between each assignment, our changing state function.

Based on the control flow graph of the operation in the SSA form, TD extracts the behaviors, from which the test targets are issued. Thus, a behavior is the conjunction of all predicates on paths starting from the root node of the graph, until the leafs, through a depth-first search in the graph. A test target is then computed as the set of states in which the behaviors can be activated.

For each test target, TD automatically computes a sequence of operations, a preamble, from the initial state of the system, that aims at its activation. This gives the test cases, to which an oracle is associated. This latter can be defined in two ways: the state of the model after test execution, and/or values returned by the operations of the test. In our framework, we only consider this second definition, as iTac-QoS does not have access to the internal states of the WS, and, moreover, the definition of a mapping relation between abstract and concrete values would be to costly in terms of effort asked to the WS vendor. As a consequence, if the results produced by the operations of the system under test are not the expected one, the test fails. Otherwise, it succeeds.

Our method has been applied on the example, as shown in (Pretre, Bouquet and Lang, 2008). When we tried to generate tests on the final model, TD found fifty-two test goals. Only twenty-three of these were about the WS, the others were about messages getters and setters. Among these twenty-three tests goals, only twelve were reached by tests. The other test goals could not be activated for two reasons: either the goal was inconsistent or the initial state is incomplete. The main problem is that some of these test goals are introduced by the model merging process. We now present how to correct the model merging process in order to filtrate these irrelevant test goals for the test generation step.
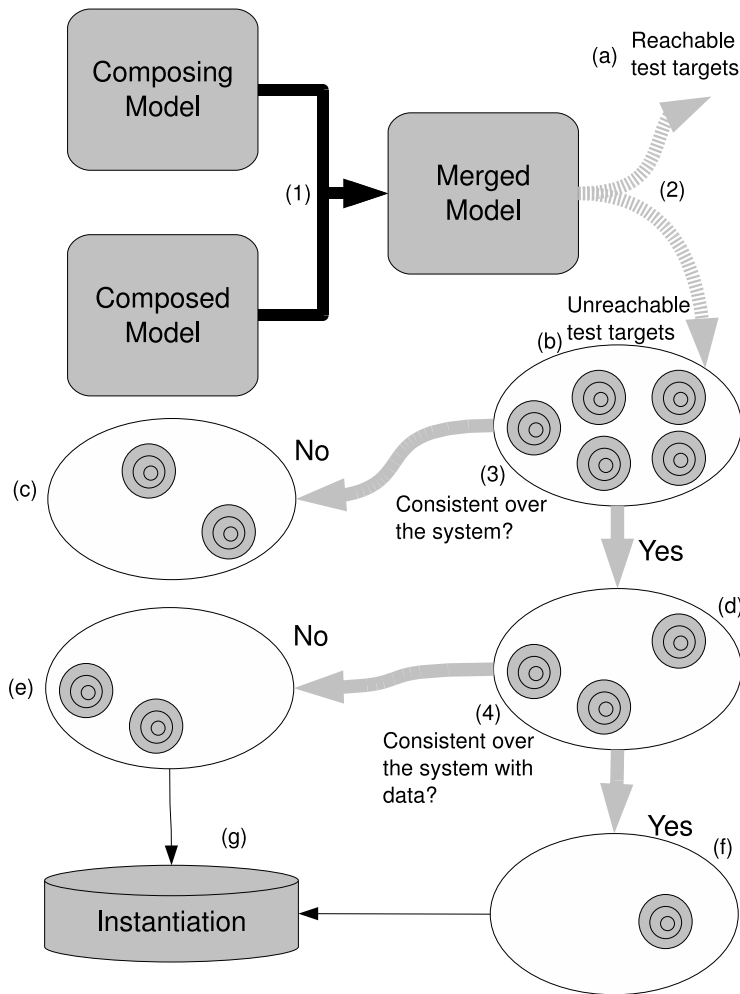
4   Correction of Model Merging

Figure 4 depicts the model correction process. The problems that we address, namely the inconsistent test targets, and the incompleteness of the initial state, are created by the model merging method.

First, we present in 4.1 our methods for checking the correctness of the test targets. Then, we give, in 4.2, an example of the behavior computation that we use

Figure 4   Correction method

for our detections. We further explain in 4.3 how to get the inconsistent behaviors and, eventually, how to get a minimal instantiation of the initial state in 4.4.

4.1   Correction Methods

The model-based testing approach of TD aims at producing tests that cover the tests targets (2). These tests are composed of sequences of operations that reach a state in which a behavior (ie. a test target) can be activated (a). If TD can not find such a sequence, the behavior is declared *unreachable* (b).

As test targets coverage is used to give an indication of the quality of the produced test suite, the fact that test targets are indifferently reported as "un-

reacheable" falsifies the results of the test generation, and may let the user think that this step did not fully succeed, whereas inconsistent targets introduce a disturbance in the statistics. This part aims at resolving this problem.

The first step of the correction (3) is to check if a behavior is inconsistent (contains contradictory predicates) over any possible configuration of the system (ie., whatever the initial state might be; without taking the initial state of the model into account). Inconsistent behaviors identified in (c) are thus excluded from the test generation process (meaning that TD will not be asked to provide tests that cover such targets). Inconsistent behaviors may appear because, in the model merging process, operation calls are substitued by copies of the entire operation, instead of using only the relevant part (a specific execution path of the called operation). Such inconsistent behavior (dead code) may only be detected by evaluating the behavior conditions.

Once we have checked that a behavior is consistent for any initial state, we refine this result by considering the remaining "consistent but unreachable" behaviors (d) and we check if the behavior is still consistent w.r.t. the initial state given in the UML model (4). If this behavior is declared consistent (f), then it means that the behavior is truely unreachable (given the limitations of the test generation tool that works in bounded time and depth for performing the state exploration). Otherwise, it means that the initial state was incorrectly instantiated, ie., the number of existing instances at the initial state was not sufficient for activating the behavior. In order to assist the modeller in correcting its model, we propose a possible instantiation of the initial state (g) that would have ensured the activability and the consistency of the considered behavior.

We now present the techniques employed for determining the consistency of the behaviors and the possible instantiation of the initial state.

4.2   Behavior Example

In order to illustrate our method of merging correction on the *validate* operation in an understable manner, we represent the operation as a control-flow graph, based on its SSA form. In this representation, shown in Figure 5, alternative branches symbolize a conditional expression in the original operation. This representation is also given by the code hereafter.
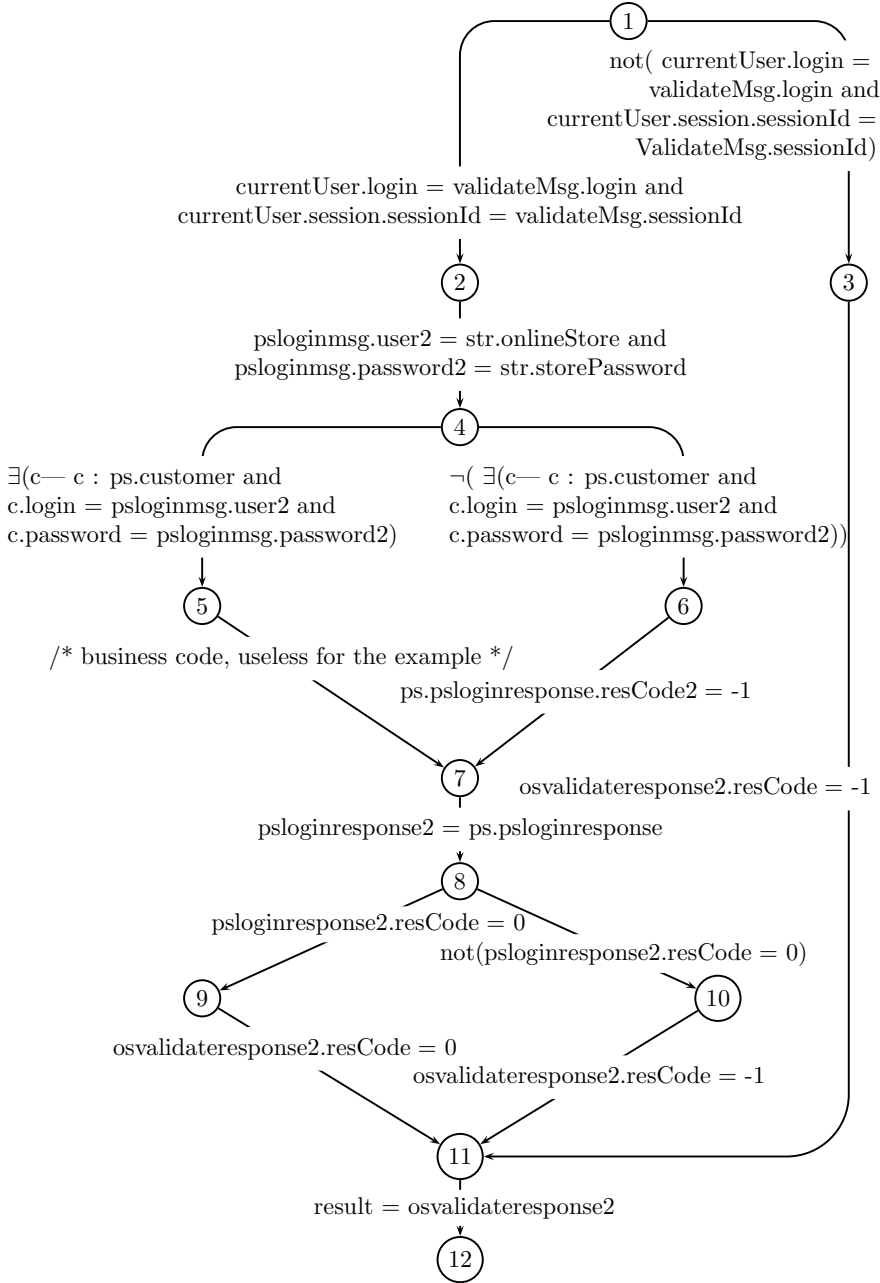
```
if (currentUser.login = validateMsg.login and
    currentUser.session.sessionId = validateMsg.sessionId) {
  psloginmsg.user2 = str.onlineStore and
  psloginmsg.password2 = str.storePassword and
  /* Code from login is brought here */
  if (exists(c| c : ps.customer and
              c.login = psloginmsg.user2 and
              c.password = psloginmsg.password2)) {
        /* business code, useless for the example */
  } else {
        ps.psloginresponse.resCode2 = -1
  }
  and
  /* end of login code*/
  psloginresponse2 = ps.psloginresponse and
  if (psloginresponse2.resCode = 0) {
    /* business code, useless for the example */
    osvalidateresponse2.resCode = 0
  } else {
    osvalidateresponse2.resCode = -1
  }
} else {
  osvalidateresponse2.resCode = -1
}
and
result = osvalidateresponse2
```

Behaviors are then computed by a depth-first search in the resulting graph. It may be seen as a DNF decomposition of the SSA representation of the operation. In this paper, we focus on the third behavior (denoted b_3), corresponding to path $(1, 2, 4, 6, 7, 8, 9, 11, 12)$, and the fourth behavior (denoted by b_4), corresponding to path $(1, 2, 4, 6, 7, 8, 10, 11, 12)$ of Figure 5.

Figure 5 *Control-flow graph of the SSA form of the* `validate` *operation*

Behavior b_3 of the login validation.
    The expression of behavior b_3 is given by the following set-theoretical predicate:

$currentUser.login = validateMsg.login \wedge$
$currentUser.session.sessionId = validateMsg.sessionId \wedge$
$psloginmsg.user2 = str.onlineStore \wedge$
$psloginmsg.password2 = str.storePassword \wedge$
$\neg (\exists c . c \in ps.customer \wedge$
$\qquad c.login = psloginmsg.user2 \wedge$
$\qquad c.password = psloginmsg.password2)) \wedge$
$ps.psloginresponse.resCode2 = -1 \wedge$
$psloginresponse2 = ps.psloginresponse \wedge$
$psloginresponse2.resCode = 0 \wedge$
$osvalidateresponse2.resCode = 0 \wedge$
$result = osvalidateresponse2$

    The set-theoretical expression of the fourth behavior b_4, is the same as for b_3 except for the last part,

$psloginresponse2.resCode = 0 \wedge osvalidateresponse2.resCode = 0 \wedge$
$result = osvalidateresponse2$

that becomes

$\neg(psloginresponse2.resCode = 0) \wedge osvalidateresponse2.resCode = -1 \wedge$
$result = osvalidateresponse2$

4.3   Consistency of Test Targets

    Behavior b_3 illustrates the problem of inconsistency of so-called "unreachable" behaviors. This latter describes a login validation, that requires the user not to be logged

```
ps.psloginresponse.resCode2 = −1
```

after a successful authentication.

```
osvalidateresponse2.resCode = 0
```

This is clearly an incoherent behavior that is introduced by the model merging process.
    The consistency checking ensures that the behavior is coherent by verifying that a proof obligation, called the behavior consistency formula (BCF), is satisfiable. The *BCF* is the conjunction of the predicates of a behavior. It represents the minimal condition to activate the behavior, in terms of system states in which the behavior may be activated.
    To be able to verify this formula using automated tools, a representation of the system state in the theory used by the prover/solver that will be employed is necessary. We have chosen to employ a set-theoretic representation of the system state. As in a previous work (Bouquet et al., 2005), the object oriented systems we consider are represented using sets, total functions and relations, according to

the following principles. The instances of the classes are represented by abstract sets of atoms (denoted by $INSTANCES\_class\ name$) that give the addresses of the possible objects for a given class. Each instance attribute is a total function (denoted by $\rightarrow$) mapping the instance to its value in a given domain. Each binary UML association is represented by a relation (denoted by $\leftrightarrow$) between the first and the second class instances. Thus, for a given instance, it is possible (using domain or image restriction of the relation) to know to which instances it is linked. An exception is done for $(0..)1 \leftrightarrow *$ associations, that are translated into total functions that associate an instance of the $(0..)1$ side (possibly a specific constant named *null* for the 0-multiplicity cases) to the instances on the $*$ side. The data model is extracted from the considered UML class diagrams. The set of existing instances and relations between instances is given by the instance diagram considered by TD.

BCF of the login validation.

A part of the BCF expression of behavior b_3 is given by the following set-theoretical predicate:

$$onlineStore \subseteq INSTANCES\_ONLINESTORE \ \wedge$$
$$parcelService \subseteq INSTANCES\_PARCELSERVICE \ \wedge$$

$$customer2 \subseteq INSTANCES\_CUSTOMER2 \ \wedge$$
$$customer2\_2\_login \in customer2 \rightarrow STR \ \wedge$$
$$customer2\_2\_password \in customer2 \rightarrow STR \ \wedge$$
$$session2 \subseteq INSTANCES\_SESSION2 \ \wedge$$
$$session2\_2\_id \in session2 \rightarrow STR \ \wedge$$
$$isLoggedIn2 \in customer2 \leftrightarrow parcelService \ \wedge$$
$$knows2 \in customer2 \rightarrow parcelService \ \wedge$$
$$isLoggedWith2 \in customer2 \leftrightarrow session2 \ \wedge$$
$$manages2 \in session2 \rightarrow parcelService \ \wedge$$
$$.... \ \wedge$$
$$BCF(b_3)$$

The *BCF* is verified using satisfiability checker tools, namely SMT-provers and a set-theoretical solver. Provers use deductive techniques that make it possible to deal with infinite data, such as abstract sets whose content and size is not specified. Constraint solver manage the constraints using consistency algorithms. Unfortunately, consistency algorithms are not complete, and thus, an enumeration of the solutions, named labeling, is potentially necessary for ensuring the satisfiability of the constraints.

In our experimentation, we use the CLPS-BZ solver (Bouquet, Legeard and Peureux, 2004) (that natively supports set theory) and the SMT provers (Ranise and Tinelli, 2006), in which the set theoretical structures are expressed using the array theory (suitable for representing sets and functions) as done in (Couchot and Dadeau, 2007) and (Couchot et al., 2003).

For the BCF of behavior b_3, the tools automatically conclude on the unsatisfiability of the formula without considering the initial state of the model. The SMT-provers Yices answers in 0.16s, Z3 in 0.14s. CLPS-BZ can not be employed because it requires finite data domains, that do not exist for this formula. It concludes on <1s if the initial state of the model is given to complete the formula.

Behavior b_4 consistency formula is declare satisfiable in 25.89s for Yices and runs out of memory for Z3. But, considering the initial state of the model, the formula is declared unsatisfiable, in 1s for CLPS-BZ, 3.74s for Yices and 38.31s for Z3.

With this method, we can automatically find inconsistent behaviors in the operations of an UML/OCL model. For the example model, 23 generate tests were generated, excluding 11 "unreachable" targets. Among these, we found 8 inconsistent behaviors (as for b_3) meaning that the remaining 3 behaviors are "real" unreachable ones for the considered initial state (as for b_4).

When verifying the *BCF* we do not take the initial state (existing instances and their links) into account. But, it might be possible that the initial proposed is not "sufficient" for the test generation process to reach certain test targets. Indeed, objects can not be created dynamically when executing the UML models (a restriction of TD). Only links between instances can be changed with adequate operations that can compose the preamble of the test cases. We now propose to check if the inconsistency of the behaviors may possibly due to an incorrect initial state that would not contain enough instances and/or incorrect links between them. The goal of this part is to find whether it is possible to find a configuration of the system (instances + links) that makes it possible to activate a given behavior.

4.4   Computation of a Relevant Initial State

As seen before, the inconsistency of behaviors may be due to an erroneous initial state of the UML instance diagram.

After having filtrated the inconsistent behaviors, we propose an instantiation of the initial instances of the system that would make it possible to activate a behavior.

Behavior b_4 illustrates the instantiation problem. This behavior describes a login validation failure, that requires the existence of a user having the adequate login/password values

```
exists(c|  c : ps.customer and
           c.login = psloginmsg.user2 and
           c.password = psloginmsg.password2))
```

whereas such an instance does not exist in the instance diagram.

Two possibilities may cause this behavior to be inconsistent w.r.t. the given initial state: either instances are missing (and since they can not be created by TD this impacts the reachability of the behavior), or the configuration of the links between instances can not be obtained from the original configuration, through the application of the model operations. We call **instantiation** this process of finding a suitable configuration for activating the behavior. This method reuses the data structure of the system expressed in the set theory. It is now described.

Data Structure Representation

As in the section 4.3 we get the *BCF* with the system representation in predicative form expressed in the set theory. All class instances sets are pairwise distinct and considered as primary sorts.

We compute the *graph of sort dependency* (Bouquet et al., 2007a), a representation of dependency between sorts. The nodes are the sorts and the edges are the dependencies. A sort s depends of others sorts $s_1, \ldots, s_n$ if there exists a functional symbol whose signature is $s1, \ldots, s_n \rightarrow s$. Intuitively it is the case when an element transformer adds some elements of some sorts in an other sort.

At this point, we obtain the data structure of the model over the set theory and their dependencies.

Instantiation of the Instance Sets

The instantiation computation gives an over-approximation of needed objects in the instance sets for the behaviors and reduce it. Therefore, we want to get each variables of the *BCF* representing a system object and put it in the appropriated sort.

We skomelize the *BCF* to extract these variables, that become constants of the Herbrand universe, and we use informations of the behavior predicate to fill the data structure with these constants.

A simple way to do this is to duplicate and add each constant in each sort. This way, we are sure that any variable can be instantiated by any sort. In the same time, we use the graph of sort dependency to complete the instantiation. For each dependency and each combination of constant which can be used to apply the function, we add a new fresh constant corresponding to the result of the function application over this combination.

The problem of this approach is that the Herbrand universe calculus can not terminate, due to a cycle in the dependency graph of sort. Indeed, the computation of the Herbrand universe is finite if and only if the graph is acyclic. Nevertheless, for an object-oriented representation, the corresponding graph of sorts is necessarily acyclic (a class can not be its one of its own subclass, and moreover, TD does not support inheritance) and thus, its computation eventually terminates.

We check the *BCF* on a solver with this upper approximation as parameter domain to reduce it and obtain a minimal behavior instantiation.

We obtain for the behavior b_4 and the $USER$ class an intantiation where a user c exists and satifies the condition

$$\exists c \mid c : ps.customer\ and\ c.login = psloginmsg.user2$$
$$and\ c.password = psloginmsg.password2$$

The solution found for this behavior is provided to the WS vendor as an illustration of an acceptable configuration that would make it possible to activate the previously-unreached behavior. The vendor is then invited to analyse this example and check if it can be integrated in his model.

5 Conclusion and future work

In this article, we introduced a method used to merge UML models for test generation. The method takes place in a quality evaluation framework for WS: iTac-QoS. In this framework, WS vendors produce a UML model of their product. Tests are generated from this model, and results of their execution are translated

into marks, allowing customers to evaluate quality of the WS.

Our framework was not able to test WS which acted as client of other WS. This was due to the fact that a WS vendor is not able to produce a valid UML model of a foreign product. We solved this problem with our model merging method.

This method is done in four steps. The first one is to transform state-charts into OCL code. Once done, a new class diagram is created, which includes classes from the several models. To avoid any overwriting due to two classes sharing the same name, classes are renamed before the creation of the new class diagram. The third step is to merge all OCL code into a single class (the one modeling the WS under test). We have to do this because the test tool we use (TD), does not handle operation call. The solution proposed to solve this problem is based on a another specificity of the test tool. OCL code is interpreted as a sequential language, not a parallel one. The last step of model merging is to create a new instance diagram.

To the best of our knowledge, this work is an innovation in model merging. We do not know any solution allowing to automatically merge UML models for test generation.

This solution has three main advantages. The first one is its portability. Except for OCL merging, it can be applied to classical UML model which are not designed for TD. The OCL merging solution only works because of an TD specificity, but has been designed to solve a lack of this software.

The other advantages are more specific to our framework. The first one is that our framework can now evaluate quality of WS performing distributed compositions. The last one is that it simplifies work for WS vendor: they just have to model their own product, and our framework automatically produce a model describing the whole composition. They do not have to specify composition pattern for merging. Composition specification is done by describing composed WS interface in the class diagram, and to call operations in OCL code.

In order to simplify description of interfaces in class diagram, we created a tool translating WSDL files in XMI (http://lifc.univ-fcomte.fr/~pretre/itac-qos/wsdl2xmi. php.html). With this tool, describing interfaces is really simple and automated.

We are currently working on a last improvement for our method: initial state completion. As shown in section 3.5, TD was not able to generate some tests on model produced by our example.

Our goal is to be able to automatically classify those test goals: do they concern inconsistent behaviour or can we add data to make this test goal activable? Once inconsistent test goals set aside, we will have to complete models to reach all test goals.

We are currently working on integrating the inconsistency detections in the model merging phase and to integrate it in TD in order to filtrate test targets, so as to improve their coverage rate.

References

Benatallah, B., Dumas, M., Fauvet, M.-C., and Rabhi, F. A. (2003) 'Towards patterns of web services composition', *Patterns and skeletons for parallel and distributed computing*, (pp.265–296).

Bertolino, A. and Polini, A. (2005) 'The audition framework for testing web services interoperability', *EUROMICRO-SEAA* (pp.134–142).

Bézivin, J. (2001) 'From object composition to model transformation with the mda' *TOOLS '01: Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS39)* (pp.350). Washington, DC, USA: IEEE Computer Society.

Boronat, A., Carsí, J. A., Ramos, I., and Letelier, P. (2007) 'Formal model merging applied to class diagram integration', *Electron. Notes Theor. Comput. Sci.*, Vol.166, pp.5–26.

Bouquet, F., Couchot, J.-F., Dadeau, F., and Giorgetti, A. (2007a). 'Instantiation of parameterized data structures for model-based testing', *Proceeding of the 7th Int. B Conference (B'2007)*, Vol. 4355, pp.96–110. Besancon, France: Springer.

Bouquet, F., Dadeau, F., Legeard, B., and Utting, M. (2005) 'Symbolic animation of JML specifications', *In:* J. Fitzgerald, I. Hayes, and A. Tarlecki (Eds.)*, Procs of the Int. Conf. on Formal Methods (FM'2005)*, Vol.3582 of *LNCS* (pp.75–90), Newcastle Upon Tyne, UK: Springer-Verlag.

Bouquet, F., Grandpierre, C., Legeard, B., Peureux, F., Vacelet, N., and Utting, M. (2007b), 'A subset of precise UML for model-based testing', *Proceeding of 3rd int. Workshop on Advances in Model Based Testing (A-MOST'07)* (pp. 95–104). London, UK: ACM Press. A-MOST'O7 is colocated with ISSTA 2007, Int. Symposium on Software Testing and Analysis.

Bouquet, F., Legeard, B., and Peureux, F. (2004) 'CLPS-B: A constraint solver to animate a B specification', *International Journal on Software Tools for Technology Transfer (STTT)*, Vol.6, No.2 pp.143–157.

Christensen, E., Curbera, F., Meredith, G., and Weerawarana, S. (2001) 'Wsdl 1.1.' http://www.w3.org/TR/wsdl.

Clarke, S. (2002) 'Extending standard uml with model composition semantics', *Sci. Comput. Program.*, Vol.44, No.1, pp.71–100.

Couchot, J.-F. and Dadeau, F. (2007). 'Guiding the Correction of Parameterized Specifications', *Integrated Formal Methods, IFM'07*, Vol.4591 of *LNCS* (pp. 176–194).

Couchot, J.-F., Déharbe, D., Giorgetti, A., and Ranise, S. (2003) 'Scalable Automated Proving and Debugging of Set-Based Specifications', *JBCS*, Vol.9, No.2, 17–36.

Estublier, J. and Ionita, A. D. (2005) 'Extending uml for model composition', *Proceedings of the 2005 Australian conference on Software Engineering (ASWEC '05)* (pp.31–38). Washington, DC, USA: IEEE Computer Society.

Foster, H., Uchitel, S., Magee, J., and Kramer, J. (2003) 'Model-based verification of web service compositions', *ase*, Vol.0, pp152. Los Alamitos, CA, USA: IEEE Computer Society

Frantzen, L., Tretmans, J., and de Vries, R. (2006) 'Towards model-based testing of web services', 1 *In* A. Bertolino and A. Polini (Eds.), *Proceedings of International Workshop on Web Services Modeling and Testing (WS-MaTe2006)* (pp.67–82). Palermo, Sicily, ITALY.

Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J.-J., Nielsen, H. F., Karmarkar, A., and Lafon, Y. (2007) 'Soap version 1.2.', http://www.w3.org/TR/soap12-part1/.

Heckel, R. and Lohmann, M. (2005) 'Towards contract-based testing of web services', *In* M. Pezzé (Ed.), *Proceedings of the International Workshop on Test and Analysis of Component Based Systems (TACoS 2004)*, Vol.116 (pp.145–156).

Heckel, R. and Mariani, L. (2005) 'Automatic conformance testing of web services', *Proceedings of the 8th International Conference on Fundamental Approaches to Software Engineering (FASE 2005)* pp.34–48.

Marcos, E., de Castro, V., and Vela, B. (2003) 'Representing web services with uml: A case study', *In* M. E. Orlowska, S. Weerawarana, M. P. Papazoglou, and J. Yang (Eds.), *ICSOC*, Vol.2910 of *LNCS* (pp.17–27).

Myers, G. J. and Sandler, C. (2004) 'The Art of Software Testing'.

OASIS UDDI specification TC (2005) 'Uddi version 3.0.2.', http://uddi.org/pubs/uddi_v3.htm.

Pretre, V., Bouquet, F., and Lang, C. (2007) 'A model-based validation framework for web services.' *Proceedings of the 5th workshop on Systems Testing and Validation (STV07)* pp.63–76.

Pretre, V., Bouquet, F., and Lang, C. (2008) 'Automating UML models merging: application of the method', *Research Report RR2008-09*, LIFC.

Ranise, S. and Tinelli, C. (2006) 'The Satisfiability Modulo Theories Library (SMT-LIB)', www.SMT-LIB.org.

Staikopoulos, A. and Bordbar, B. (2005) 'A comparative study of metamodel integration and interoperability in uml and web services', *In* A. Hartman and D. Kreische (Eds.), *ECMDA-FA*, Vol.3748 of *LNCS* pp.145–159.

Thöne, S., Depke, R., and Engels, G. (2002) 'Process-oriented, flexible composition of web services with uml', *In* S. Spaccapietra, S. T. March, and Y. Kambayashi (Eds.), *ER (Workshops)*, Vol.2503 of LNCS pp.390–401.

Vogels, W. (2003) 'Web services are not distributed objects', *IEEE Internet Computing*, Vol.7, No.6, pp.59–66.

Eid, M., Alamri, A. and El Saddik, A. (2008) 'A reference model for dynamic web service composition systems', *Int. J. of Web and Grid Services*, Vol.4, No.2, pp.149–168.