Available and Reliable Services

J.L. Pastrana, E.Pimentel
Dpto. de Lenguajes y Ciencias de la Computación
University of Málaga. Málaga, Spain
{pastrana,ernesto}@lcc.uma.es

M. Katrib
Departamento de Computación
University of Havana. Havana, Cuba
mkm@matcom.uh.cu

***Abstract*—This paper presents a framework and a tool for reliable and available services composition and coordination based on connectors defined by service client and automatically generated by the COMPOSITOR tool we have developed. Connectors use contracts to express the non-functional requirements and the behavior desired by the client of a service, such as QoS (Quality of Service) features. The connectors generated are self adaptive. The adaptation enactment is based on using an OWL ontology of the server domain which is used to adapt any mismatch when invoking a service at run-time if the server is updated or replaced. This makes services reliable and available.**

Keywords-component; Services; self-adaptation; Availability; Reliability

I. INTRODUCTION

Self-adaptive software systems [1,2] are those able to manage changing operating conditions dynamically and autonomously. Such self-adaptive systems can configure and reconfigure themselves, augment their functionality, continually optimize themselves, protect themselves, and recover themselves, while keeping most of their complexity hidden from the user and administrator. Currently, most proposals in this field rely on an explicit representation of the components and goals of the system (usually following a top-down approach), or on the definition of local rules for the different elements of the system, which results in an emergent self-organizing behavior (hence following a bottom-up approach). Both these approaches are suitable for closed systems, that is, those whose constituent components are well known at design time, or where there is no need to explicitly represent the goals of the system.

However, there are many situations in which software systems are open, and the changes in their execution environment are directly subject to the availability of a particular service, which may join or leave the context of the system at any given moment. These systems lack a predefined description of their architecture and components, and even of their goals. In such open systems, new adaptability problems arise, such as the connection and disconnection of a new software component to an already running system; or how to solve interoperability issues among third-party components not specifically designed to interact with each other.

The current proposal presents a framework (including a methodology supported by a tool) based on a notion of connectors that allows using and composing Web Services. It also allows the non-functional requirements of a service client to be set. This way the user of a service can set the behavior and QoS features (performance, security, network reliability, etc.) he/she expects from it. Moreover, connectors use one ontology of the server domain to provide self-adaptation, so the system developed will work properly even if a new Web Service comes into the system, or the current Web Service is updated.

The current proposal takes into account a development methodology associated to connectors, supports QoS including specific QoS-related primitives such as *Authorized*, *TimeOut* and *AverageResponseTime*, presents a new predefined primitive to express non-functional properties (*Ignore*) and improves the adaptive features of connectors allowing it to discover and connect to a new server using primitives such as *connectTo*, *SetUDDIServer* and *SetListServers*, as well as, adapting a service as a composition of some services and solving data conversion problems. Finally, some benchmark tests have been taken in order to evaluate the cost of using connectors and the cost of the adaptation process.

The case study of a library application where users can load/upload papers has been selected, in order to explain the different concepts with a consistent reference scenario that runs through the whole paper. Basically, the application will use an external Web Service for papers storage and management and both work together in a loosely coupled way. This case study will be used to show how using our framework, the non-functional and QoS requirements of a client can be expressed using Meyer design by contract style (for example, suspending a call to synchronize, ignoring a request and returning a new value, null object refactoring, changing the server used when the average responding time is greater than wanted, etc.), and how the framework can automatically solve the run-time adaptation problems caused when the server is updated or replaced by other semantically equivalent.

II. THE FRAMEWORK

One of the main goals of Web Services is reusability, and, one of the essential tasks in service-based software development is finding the right Web Service providing the functionality and interface required by component clients. Once the services are identified, a connector can be used to adapt the behavior and to set the nonfunctional requirements desired of the Web Service, without modifying the services themselves. The steps for this adaptation involve the definition or selection of one of the existing ontologies of the domain of the server, setting the invariant wanted for the Web Service and what to do when it fails as well as setting the precondition and the postcondition wanted for each service and what to do when they fail.

A connector is itself a Web Service that mediates between the client component and the server. Hence, the connector mechanism does not try to change any internal part of the server. A connector is a component for managing, connecting and adapting the behavior of a server component for any client. All the constraints imposed on the server are established as contracts (preconditions, postconditions and invariants) represented by logical expressions.

The connector checks the invariant and the preconditions for each service before calling it and checks the postconditions and the invariant when the server returns the control. It can also

execute some defined actions when the contract fails. This way, a client is able to set how he/she wants the server to behave and he/she is also able to set what should be done when the requirements fail (Figure.1).

Connectors present the following features:

- Defined by clients and implemented as a Web Service.
- Using contracts to express the QoS properties and non-functional behavior s expected by clients.
- Automatically generated.
- Allowing clients to work with one or more different servers obtaining the same behavior.
- Allowing clients to use a server as it was a monitor.
- Enriched with server domain ontology in order to adapt requests.
- Allowing extension techniques (inheritance, delegation and composition).

These features improve the traditional concept of connector. Using design by contract, the connectors will not be just a type or interface adaptor. They allow the client to define under which conditions they wish the service to be executed and what to do when a contract (precondition, postcondition or invariant) fails. Connectors are active components, so they have an internal state which makes possible to do actions before and after the call (for example, to suspend a call until its precondition be satisfied).

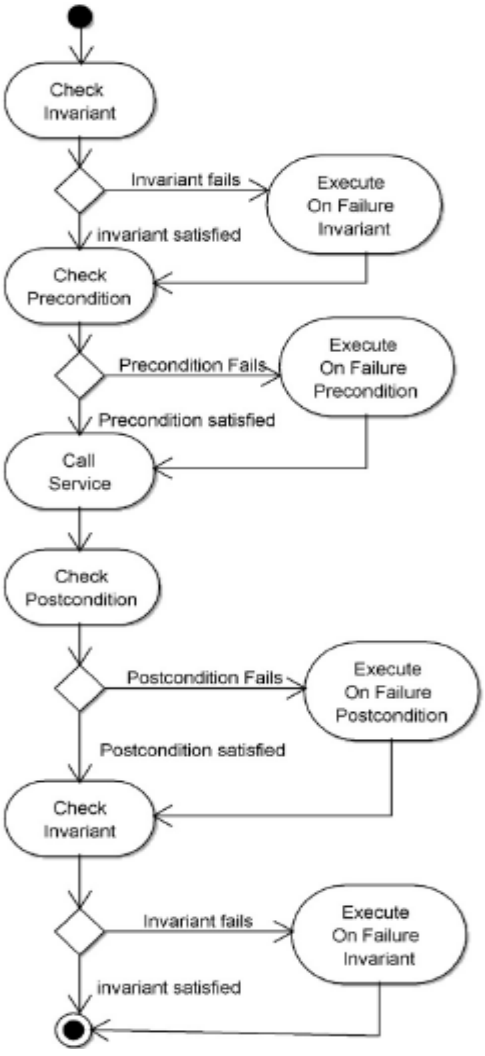

Figure 1. Connector Model Activity Diagram

Moreover, contracts allow non-functional requirements to be expressed as synchronization or timeouts, as well as QoS properties for each service.

Finally, connectors are able to achieve dynamic adaptation. They use an ontology and an inference machine (Prolog) to adapt not found requests when the server has changed or been updated. The run-time adaptation process is executed when the connector tries to call a service and gets a ‘‘not found service’’ exception from the server. Then, the connector gets by reflection all the services provided by the server and using a first order predicate logic engine tries to unify the requested service with the knowledge expressed in the OWL ontology provided to the connector which has been translated to Prolog Horn clauses. Every class defined in the ontology is translated to an *owlClass* clause, every property is translated to an *objectProperty* clause and every subclass, every “same as” relationship and every equivalence relationship defined in the ontology are translated to *owlSubClassOf*, *sameAs* and *equivalentClass* clauses.

We will present now the Prolog rules used to adapt a request. Just only the main clauses will be detailed where the definition of some auxiliary clauses could be assumed.

```
canBeReplaced(C1,S1,P1,C2,S2,P2):-
    owlClass(C1), canBeReplacedClass(C1,C2),
    objectProperty(C1,S1,P1),
    canBeReplacedProperty(S1,S2),
    equivalentParam(P1,P2).
```

This rule means that a call to a service *S1* of the server *C1* with the parameters *P1* can be replaced by a call to a service *S2* of the server *C2* with the parameters *P2* if the server *C1* and the service *P1* are defined in the ontology and *C1* can replaced by *C2*, *S1* by *S2* and the parameters *P1* and *P2* are equivalents.

To replace a server by other we use the following rules:

```
canBeReplacedClass(C,C).
canBeReplacedClass(C1,C2):- owlSubClassOf(C2,C1).
canBeReplacedClass(C1,C2):- sameAsReflexive(C1,C2).
canBeReplacedClass(C1,C2):-
        equivalentClassReflexive(C1,C2).
```

These rules mean a server can be replaced by itself, by a subclass or by a server which is defined in the ontology as the same or equivalent. In addition, to replace one service by another we use the following rules:

```
canBeReplacedProperty(S,S).
canBeReplacedProperty(S1,S2):-
        sameAsReflexive(S1,S2).
canBeReplacedProperty(S1,S2):-
        equivalentPropertyReflexive(S1,S2).
```

These rules mean a service can be replaced by itself, or by a service which is defined in the ontology as the same (same functionality) or equivalent (equivalent functionality). Finally, to replace parameters by others we use the following rules:

```
equivalentParam([],[]).
equivalentParam([XjR1],[XjR2]):- owlClass(X),
        equivalentParam(R1,R2).
equivalentParam([XjR1],[YjR2]):- owlClass(X),
        owlClass(Y),canBeReplacedClass(X,Y),
        equivalentParam(R1,R2).
equivalentParam(P1,P2):- permutation(P2,LP2),
equivalentParam(P1, LP2)
```

These rules mean that two sets of parameters are equivalent if they are empty or they are the same or each individual parameter from *P1* can be replaced by its corresponding one in

P2 in the same order or a permutation. In order to express parameter type conversion Prolog functions can be used and it is possible, as well, to write a clause to specify that a service can be equivalent to a sequence of services. For example, you can express the type conversion from *C2* to *C1* through the function *C2ToC1* and the equivalence between *S1* and the composition of the services *S2* and *S3* through the predefined function composition as follows:

equivalentClass(C1, C2ToC1(C2)).
equivalentProperty(S1, composition([S2,S3])).

Users may use contracts to define their own non-functional requirements as well as the behavior to be executed when contracts fail. Because of a connector is also implemented as a Web Service using C# and ASP.NET, preconditions, postconditions and invariants will be valid Boolean expressions written in C#. The behavior s associated to each contract are C# sentences. Both, contracts and behavior s can use a number of predefined predicates and functions that model nonfunctional or QoS behaviors. In addition, other Web Services (or connectors) that are going to be used in the context of the application can be declared using external sentences to be used as part of the contracts. It is worth noting that a connector can be used by different clients in order to obtain the same behavior of a server, as well as using another (or new) connector to get a different behavior of the same server.

QoS properties can be defined as part of the contracts in the connectors. Our proposal offers predefined functions for performance (execution time) and security only because other features are implicit in the model or its implementation (scalability, capacity, availability, robustness, accessibility and interoperability).

- Performance: It is possible to use some performance metrics in the invariant or preconditions such as response time (*AverageResponseTime* predefined function) or completion time (using *Timeout*). For example, it is possible to set as an invariant of a Web Service that the average response time of any service will be less than a particular time. This way, the connector will verify this property before and after a service is invoked.
- Scalability: The delegation extension technique for connectors allows connectors to be scalable because it is a way to add new functionalities to the connector.
- Capacity: A connector is a multi-thread Web Service; so many concurrent requests can be served.
- Reliability, availability and robustness: Self-adaptation makes connectors robust and fault tolerant.
- Accessibility and interoperability: Connectors are implemented as Web Services and Web Services provide seamless connections from one software application to another because they use standard protocols (XML, http, SOAP).
- Security: Security can be managed using the *Authorized* predefined function. This primitive verifies if a client is authorized to call the service using digital signatures.

In order to promote effective reuse of connectors, the proposed approach allows three different mechanisms for extending connectors: subtyping (inheritance), delegation and composition. Firstly, inheritance allows us not only to reuse the same behavior for the server but also to improve the conditions we need for those services (a weaker precondition and/or a stronger postcondition). However, connector inheritance is not allowed to add functionality to the connector. Delegation is a powerful tool for adding functionality to the connector statically as well as dynamically (at run-time). Finally, the composition allows us to implement filter techniques to check and solves behavior s step by step. In the following sections, these techniques will be analyzed and how they can be used for connectors will be shown.

III. THE STUDY CASE

This section will show some features of the connectors via a simple but easily understandable example. Let us suppose we want to develop our study case. We have found a paper database server (a Web Service) which offers the desired functionality. Following, we will show how to express some non-functional behavior s and QoS properties:

A. *Using suspend to synchronize*

Suppose we want to establish that only a registered user could send a paper. If the user is not registered we are going to send a register request and wait until he/she is registered before executing the request. We could set the user being registered as a precondition of the service *SendPaper* in the connector and when it fails the connector will launch the registration process and it will suspend the *SendPaper* service until the user is registered.

```
void SendPaper(string user, PaperInfo info, string text)
require:      Registered(user)
on failure:   Register(user);
              Suspend();
```

B. *Ignoring a request and returning a new value*

Now let us suppose we want unregistered users to be able to obtain only the abstract of a paper. The GetPaperById request from an unregistered user will be ignored and the abstract of the paper will be returned. It could be expressed in the connector as a precondition for the GetPaperById service.

```
string        GetPaperById(string user, int id)
require:      Registered(user)
on failure:   _result = GetAbstractById(id);
              Ignore();
```

C. *Null object refactoring*

Usually, when a client calls a method on a field or variable that is null, an exception may be raised, a system may crash, or similar problems may occur. The null object pattern [3] provides a solution to such problems. Instead of assigning a null value to a reference-type property, a nullable object can be used to represent a null value, instead of directly assigning a null value itself. This makes it easier to reference and means users need not to worry about getting a null exception when using the property. Here, we want to return an empty string instead of null when looking for an abstract for which there is no corresponding name.

```
string GetAbstractByTitle(string title)
ensure: (_result!= null)
on failure: _result = "";
```

Alternatively, another possibility would be to return ‘‘title not found’’ instead of null when looking for an abstract for

```
which there is no corresponding name.
string GetAbstractByTitle(string title)
ensure: (_result!= null)
on failure: _result = title + " not found";
```

D. *Setting QoS features*

Let us suppose we want to add some QoS features to our system. For example, we want the average response time to be less than 2.5s, and, we will change the server if the average

response time is greater. Note that the new server can be set ‘‘ad hoc’’ or obtained from an UDDI server or from a list of servers using the *getNewServer* function which it will ensure that the invariant will be satisfied.

As the invariant is always verified before and after a service is called and the AverageRespondTime tells the average time taken by the call it is possible to set the following invariant for the connector.

invariant: AverageResponseTime()<2.5
on failure: ConnectTo(getNewServer())

E. Run-time adaptation

Suppose the server managed by the connector has been changed by a *ConnectTo* function or it has been changed or updated by another team. The original server used only one call (*SendPaper*) for sending a paper and its abstract.

However, this new server needs two calls for sending a paper. First, the abstract must be sent (*SendAbstract*) and later the paper must be sent (*SendPaperText*). This adaptation is done automatically by the connector, as can be seen in the sequence diagram shown in Figure 2, using the information in the ontology which in effect means that the composition of *SendAbstract* and *SendPaperText* is equivalent to *SendPaper*. This information is translated to a Prolog clause as follows:

```
equivalentProperty(sendPaper,
        composition([sendAbstract, sendPaperText])).
```

This way, the connector (using the knowledge enclosed in the ontology and a Prolog engine) will unify the call to the service *SendPaper* with the composition of the calls *SendAbstract* and *SendPaperText* doing the adaptation.

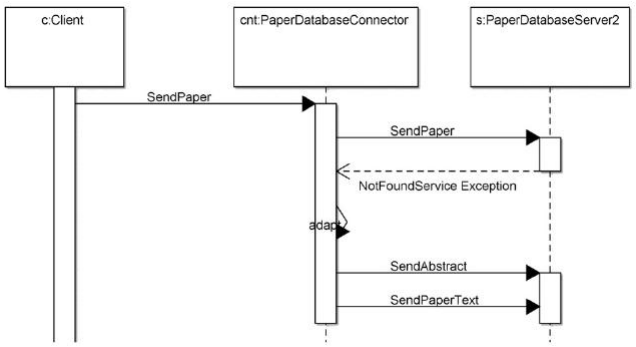

Figure 2. Adaptation sequence diagram.

IV. IMPLEMENTATION

Connectors follow the reflective middleware model [4]. In the reflective model, middleware is implemented as a collection of components that can be configured and reconfigured by the application. The middleware interface remains unchanged and may support applications developed for traditional middleware. In addition, system and application code may inspect the internal configuration of the middleware and, if needed, reconfigure it to adapt to changes in the environment through metainterfaces. In this manner; it is possible to select networking protocols, security policies, encoding algorithms, and various other mechanisms to optimize system performance for specific and often unpredictable contexts and situations.

Connectors are implemented as Web Services where external dependencies are references to remote components that are created when the connector is instantiated and they remain active while the connector is active. Every connector extends the *CConnector* class (Figure 3) that manages the predefined behavior s, the contracts and the adaptation process.

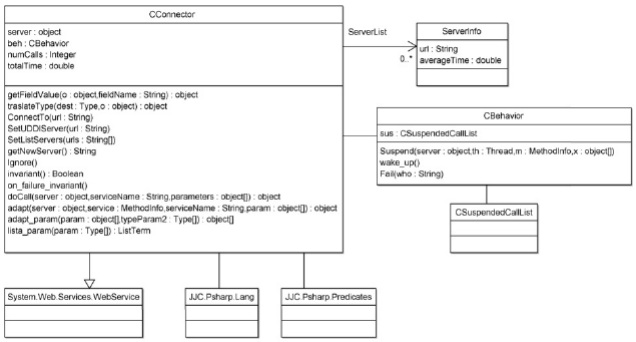

Figure 3. CConector class diagram.

Below some implementation details will be shown using the previous example. First, when the invariant is defined, a method for the invariant and *on_failure_invariant* are generated overriding the predefined invariant and *on_failure_invariant* implemented in the *CConnector* class:

This is the method which implements the invariant.

```
public override bool invariant()
{
        bool res;
        res=AverageResponseTime()<2.5;
        return res;
}
```

This method implements what must be done when the invariant fails.

```
public override void on_failure_invariant()
{
        ConnectTo(getNewServer("invariant",null));
}
```

In all, five methods are generated (precondition, postcondition, service, on_failure_precondition and on_failure_postcondition) for each service in the server. Below we show some of the methods generated for the *SendPaper* and *GetPaperById* services:

SendPaper service (as every call to a service) is managed by the method *doCall* implemented in the *CConnector* class.

```
[WebMethod]
public void SendPaper(string user, int id, string text)
{       object[] parameters = new object[3];
        parameters[0] = user;
        parameters[1] = id;
        parameters[2] = text;
        this.doCall(server,"SendPaper",parameters);
}
```

The *SendPaper* precondition (as every precondition) is generated as a bool method.

```
public bool SendPaper_precondition(string user, int id,
                string text)
{       bool res;
        res = Registered(user);
        return res;
}
```

The *SendPaper* on failure behavior is generated as a void method which calls the Register service and then it calls the

predefined Suspend behavior which is implemented by the private object *beh* declared in the *CConnector* class.

```
public void SendPaper_on_failure_precondition(
            string user, int id, string text)
{    object[] parameters = new object[3];
     parameters[0] = user;
     parameters[1] = id;
     parameters[2] = text;
     Register(user);
     beh.Suspend(this,Thread.CurrentThread,
         this.GetType().GetMethod(
         "SendPaper_precondition"), parameters);
}
```

The *GetPaperById* on failure behavior is generated as a void method which assigns as its returning value the call to the *GetAbstractById* service and calls the predefined *Ignore* behavior to ignore the current call to *GetPaperById*.

```
public void GetPaperById_on_failure_precondition(
            string user,  int id )
{
     object[] parameters = new object[2];
     parameters[0] = user;
     parameters[1] = id;
     _result = GetAbstractById(id);
     Ignore();
}
```

As mentioned before, every service is managed by the *doCall* method implemented in the *CConnector* class. This method implements the activity model shown in Figure 1. It checks the invariant, precondition and postcondition and calls its corresponding *on_failure* method when any of them fails. In addition, it measures the time to complete a service which will be used for the *AverageResponseTime* implementation. This method calls to the *adapt* method which will adapt the service request when it is necessary. In the following we show the *doCall* implementation.

```
public object doCall(object server, string serviceName,
            object[] parameters)
{
// 1
     Stopwatch time = new Stopwatch();
     Type remoteType = server.GetType();
     Type connectorType = this.GetType();
     MethodInfo remoterService =
            remoteType.GetMethod(serviceName);
     MethodInfo preService =
            connectorType.GetMethod(serviceName+
            "_precondition");
     MethodInfo postService =
            connectorType.GetMethod(serviceName+
            "_postcondition");
     MethodInfo on_failure_pre =
            connectorType.GetMethod(serviceName+
            "_on_failure_precondition");
     MethodInfo on_failure_post=
         connectorType.GetMethod(serviceName+
         "_on_failure_postcondition");
// 2
     ignore = false;
     if (!invariant()) on_failure_invariant();
     if (!(bool)preService.Invoke(this,parameters))
            on_failure_pre.Invoke(this,parameters);
     if (!ignore)
     {
            if (availableFlag) Monitor.Enter(server);
            time.Start();
            _result = adapt(server,remoterService,
                  serviceName,parameters);
            time.Stop();
            ++numCalls;
            totalTime+ =(time.Elapsed.Hours*3600.0
            + time.Elapsed.Minutes*60.0
            + time.Elapsed.Seconds
            + time.Elapsed.Milliseconds /1000.0);
            if (availableFlag) Monitor.Exit(server);
     }
     if (!(bool)postService.Invoke(this,parameters))
            on_failure_post.Invoke(this,parameters);
     if (!invariant()) on_failure_invariant();
// 3
     beh.wake_up();
     return _result;
}
```

In 1 the service and the methods that implement contracts and on failure blocks are built by reflection. In addition, a *Stopwatch* variable is created for time measurement.

In 2 the contracts are checked. Then, if the call is not to be ignored (the ignore flag is set by the *Ignore()* predefined function), it is invoked through the adapt method and the time taken is added to the *totalTime* and the number of calls served is increased. This will be used to calculate the *AverageResponseTime* as the sum of the time taken for all the requests (*totalTime*) divided by the number of calls served (*numCalls*). Note that the adaptation time (when it is necessary) will be added to the total time.

In 3, the local object *beh* that manages the predefined behavior s calls to the *wake_up()* method to wake up the oldest call suspended that verifies its precondition.

The adapt method (as can be seen in Figure 4) tries to call the requested service and when the call fails it gets by reflection all the services provided by the server and uses a Prolog engine to unify the request and the services offered using the semantic information contained in the ontology which has been translated to Prolog clauses.

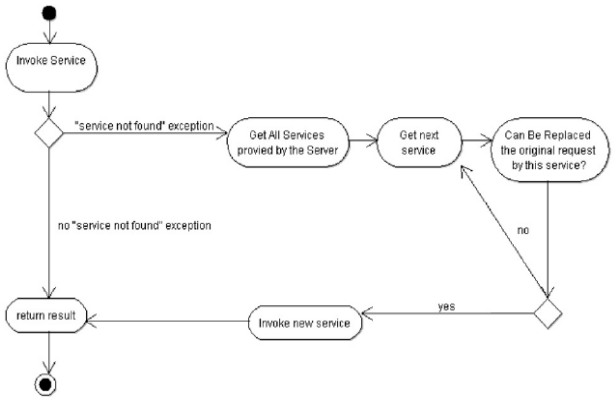

Figure 4. Adaptation model activity diagram

In the following we show a bit of its implementation.

```
public object adapt(object server, MethodInfo service,
string serviceName, object[]param)
{     //1
     try{ return service.Invoke(server,param);}
```

```
catch (Exception)
{       . . .
//2
  foreach (MethodInfo m in serverType.GetMethods())
  {
    PrologInterface sharp = new PrologInterface();
    sharp.AddAssembly(System.Reflection.Assembly.
    GetExecutingAssembly());
      . . .
    sharp.SetPredicate(new Can_Be_Replaced_6
       (C, S1, P1, C, S2, P2, new ReturnCs(sharp)));
    ok = sharp.Call();
//3
  if (ok) return m.Invoke(server,
    adapt_param(param, typeParam2));
  }
}
return null;
}
```

In 1 the method tries to invoke the service normally.

In 2 the method gets all the methods provided by the server and looks for a method in the server that can replace the service requested using the rules described before.

In 3 when the service requested can be replaced by the service m, then it is invoked and its result is returned. Data conversion problems are solved by the *traslateType* function (its activity diagram model can be seen in Figure 5) which is called by *adapt_param*.

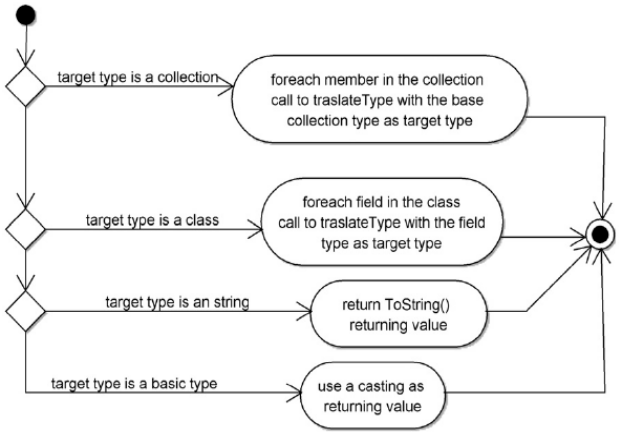

Figure 5. Data conversion model activity diagram.

V. CONCLUSIONS AND FUTURE WORK

Predictability and correctness are two key properties once the connectivity between services has been established. On the extremes, we find pure connectivity (WSDL) and pure implementation (BPEL). We need a model capable of specifying behavior of services and automatic verification of the crucial properties of their composition.

The work presented in this article deals with the need to specify the behavior of services, QoS and self-adaptation properties in Web Service based systems. Two main contributions have been provided in this article. One is the approach to managing architectural self-adaptation in the connectors (middleware level) using the knowledge of the domain (OWL ontologies). The other contribution is based on the idea that it is the client of a Web Service who sets the non-functional behavior s and QoS properties increasing the possibility of component reuse. This work improves the previous one [5] by providing a development methodology associated to connectors, supporting QoS, presenting new predefined primitives and improving the adaptive features of connectors.

Some benchmark test was done in order to evaluate the cost of connector adaptation and data type conversion. The test is based on the implementation of two Web Services offering two services each one. The first Web Service (WS1) offers one service that returns an array of 100 elements of type C1 (read) and the other one receives a class C1 element as parameter (write). The second Web Service (WS2) has the same functionality, however, the reading service has a different name (declared as equivalent in the ontology) and the elements read/written are from the class C2 (declared as equivalent in the ontology). It has been done 10, 100, 1000 and 10 000 calls. Table 1 shows the average, variance, standard deviation, maximum and minimum time (in milliseconds) for one call in the 10 000 calls test.

TABLE I. BENCHMARK TEST. TIME IN MILLISECONDS.

10 000 calls done	Avg.	Var	S. dev.	Max.	Min.
WS1 reads 100 records	3.79	76.56	8.75	312.50	0.00
WS1 write 1 record	3.22	57.13	7.56	187.50	0.00
cnt-WS1 reads 100 records	9.56	109.90	10.48	296.88	0.00
cnt-WS1 writes 1 record	6.90	140.43	11.85	359.38	0.00
cnt-WS2 reads 100 records	9.60	114.54	10.70	328.13	0.00
cnt-WS2 writes 1 record	6.67	95.96	9.80	234.38	0.00

As computing time and networking time can be considered as constant, it can be seen in Table 1 that in the worst case (it was necessary to adapt the name of the service, the type of the returning value and translate one array of 100 elements of type C1 to one array of 100 elements of type C2) the average time was increased only in 6ms what it is acceptable for non real-time systems.

As future work we will attempt to increase the number of predefined function in order to express more QoS properties (average time required to perform a service, average time to adapt a service, throughput, bandwidth, etc.) and to improve the connector generator (making it more user friendly). In addition, we intend to include in the framework a set of already developed connectors implementing QoS and synchronization patterns that can be used to generate new domain specific connectors.

REFERENCES

[1] Camara J, Canal C, Sala ¨un G. Behavioural self-adaptation of services in ubiquitous computing environments. In: Proceeding of ICSE workshop on software engineering for adaptive and self-managing systems, 2009. p. 28–37.

[2] Cheng BHC, et al. Software engineering for self-adaptive systems, URL:http://drops.dagstuhl.de/volltexte/2008/1501/pdf/08031_abstractscollection.1501.pdfS

[3] Fowler M. Refactoring. Improving the design of existing code. Addison-Wesley; 1999.

[4] Kon F, Costa F, Blair G, Campbell RH. The case for reflective middleware. Communications of the ACM. Special Issue: Adaptive Middleware 2002;45:33–8..

[5] Pastrana JL, Pimentel E, Katrib M. Composition of self-adapting components for customizable systems. The Computer Journal 2008;51:481–96.