

Smart Monitors for Composed Services

Luciano Baresi
bares@elet.polimi.it

Carlo Ghezzi
carlo.ghezzi@polimi.it

Sam Guinea
guinea@elet.polimi.it

Politecnico di Milano
Dipartimento di Elettronica e Informazione

ABSTRACT

Service-based approaches are widely used to integrate heterogeneous systems. Web services allow for the definition of highly dynamic systems where components (services) can be discovered and QoS parameters negotiated at run-time. This justifies the need for monitoring service compositions at run-time. Research on this issue, however, is still in its infancy.

We investigate how to monitor dynamic service compositions with respect to contracts expressed via assertions on services. Dynamic compositions are represented as BPEL processes which can be monitored at run-time to check whether individual services comply with their contracts. Monitors can be automatically defined as additional services and linked to the service composition.

We present two alternative implementations of our monitoring approach: one based on late-binding and reflection and the other based on a standard assertion system. The two implementations are exemplified on a case study.

Keywords

Web Services, BPEL, quality of service, functional requirements, pre- and post-conditions, composition, monitoring, .NET, XlinkIt, exception handling

1. INTRODUCTION

Web services allow for the definition of highly dynamic software systems whose components (services) can be discovered at run-time and QoS parameters negotiated during enactment. WSDL provides components with a standard interface and UDDI registries act as agencies which advertise services and allow clients to select the features needed to execute their business processes. The freedom and flexibility of such an approach, where components only interact through XML data, still needs to be thoroughly investigated.

Web services are composed in order to offer more complex services. Service composition languages include BPEL [4], where a single process orchestrates interacting services, and

WSCI [19] which supports a more decentralized composition with the different services that describe their role in the overall choreography. These compositions are often decorated with information about the QoS negotiated between the client, i.e., the business process, and the providers. Proposals like SLA [12], WSOL [20], and WS-Policy [3] move towards a more semantic composition of services and help the designer embed non-functional requirements (quality parameters) in the cooperation.

These approaches mainly concentrate on modeling the business process in terms of interacting Web services, but do not consider the behavior of such models at run-time. They do not consider how to enforce or simply probe the constraints on the composition. This is the key problem in our approach. We envision a dynamic situation where compositions are formed by assembling components selected from a serviceful environment at run-time. Selection is accomplished by matching the specifications of requested operations with the interfaces published by available services. Assertions are used to specify the behaviors¹ of both required and offered operations. Their matches define the contracts between requestors and providers.

In this paper we do not address the problem of establishing the binding between service requests and offers. We concentrate instead on how to monitor conformance between the specified service behavior and the actual service selected by the matching mechanism that establishes such binding. Such conformance can only be checked dynamically since we assume that actual services can leave or join the environment in a fully unpredictable fashion.

To investigate the monitoring problem we assume that bound compositions are described as BPEL processes. We illustrate a solution that allows the services invoked by a BPEL process to be dynamically monitored for conformance. As we mentioned, specifications are given in terms of predicate logic as assertions (mainly pre- and post-conditions).

Our proposal recalls the use of assertion languages, such as Anna [8] and APP [7], which were proposed to specify constraints via program annotations. The critical difference that makes this approach more interesting in SOA scenarios is that we are called to engage ephemeral configurations where reasoning on systems as a whole is more difficult and where, in many cases, no stake-holder has the knowledge to do so. In our case, annotations are stated as comments in the source BPEL program and then translated to generate a target monitored BPEL program. In addition to monitor-

¹Service requests should also specify non functional properties. These are ignored here for simplicity.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICSOC'04, November 15–19, 2004, New York, New York, USA.
Copyright 2004 ACM 1-58113-871-7/04/0011 ...\$5.00.

ing functional requirements, we also monitor timeouts and runtime errors in the use of remote services. Whenever any of the monitored conditions indicates misbehavior, suitable exception handling code in the generated BPEL program handles them.

The paper presents two different implementations for the monitoring service. The first approach exploits an object-oriented language and its late binding and reflection capabilities. The second implementation wraps *Xlinkit*[5], an assertion language processor, to turn it into an external service. The main differences between the two proposed implementations lie in the nature of the assertions that can be monitored. The first approach offers all the features of a programming language to specify assertions. For example, assertions can embed special-purpose methods (like the *virtual text* in Anna) or interact with external repositories to retrieve useful data. The second approach is more on the specification side and offers constructs, like existential and universal quantifiers, that are not usually supplied by programming languages.

The potential of both implementations, along with their monitoring capabilities, are exemplified on an example taken from the world of IT certification. In this example, we do not stress the dynamic aspects associated with the approach, but we concentrate on the monitoring features. External providers (services) and different stake-holders impose monitoring even in fully static systems. Moreover, every time we want to test such distributed systems we need suitable oracles to assess their correctness. This means that we foresee monitors as mandatory components in any service-based system, but the more the system embeds dynamic features and supports ephemeral behaviors, the more important monitors become.

The paper's main concern is to study the feasibility of the approach and the validity of the two implementations. These two implementations have been chosen in order to understand the strengths, the weaknesses and the differences between the two in their use for the definition of assertions in an open-world scenario. The paper is organized as follows. Section 2 introduces our approach to the monitoring problem. Section 3 describes the two proposed implementations and Section 4 exemplifies them on a case study. Section 5 discusses related proposals and Section 6 concludes the paper and indicates the directions of our future work.

2. APPROACH

Our approach provides three main mechanisms to monitor service compositions defined by BPEL programs. These mechanisms correspond to three classes of undesirable behaviors: timeouts, runtime errors and violations of functional contracts. These represent three kinds of service behaviors that can be stated using suitable *contracts* and may request suitable reaction on the service orchestration side. We express contracts for composed web services using assertions. A non intrusive way of adding assertions is to annotate our BPEL process by inserting them in the form of comments². In this way, the BPEL process remains standard in its definition and executable by any standard BPEL

²They also could have easily been encoded using the standard BPEL extensibility mechanism; we have chosen to prefer a simpler and more straightforward presentation which helps distinguish between business logic and monitoring con-

engine. This means that the first step (Fig. 1) in designing a monitored process is to design a standard unmonitored process. This can easily be done by using either textual or graphical editors that are widely available on the market. For example, both Collaxa[6] and IBM alphaWorks[11] supply graphical process editors for BPEL. After a standard process has been produced, the designer can annotate it with comments representing the contracts he or she wishes to define on timeouts, the desired behavior of the process in presence of errors, and functional contracts in terms of pre- and post-conditions.

These comments represent the core of the design overhead necessary for monitoring processes. In fact, these comments are automatically translated into the sequence of BPEL activities that augment the original process in order to make it monitored. From the designer's point of view the impact of this approach is minimal since, once the standard process and the assertions are written, the rest of the process can be completed automatically. A suitable pre-processor transforms them into BPEL code.

Contracts are translated in different ways: Timeouts and errors in service implementations can be handled using standard BPEL and good design patterns (Sections 2.1 and 2.2); functional contracts (pre- and post-conditions) require dedicated monitors (Section 2.3). As we mentioned, functional contracts are monitored by special-purpose components called **monitors**. The monitor is itself a web service and consequently becomes a part of the monitored composition. The translation of the original BPEL process into a monitored process introduces modifications to its structure. Regarding this translation, we assume that inconstances between the unmonitored process and the directives specified in the added comments must be resolved by the designer of the contracts. The monitored process reacts to misbehaviors by terminating the execution and signalling the detected problem.

2.1 Monitoring timeouts

BPEL allows the designer to associate scopes with a timeout: All the operations contained in the scope must finish their execution within the time frame set by the timeout. By defining an alarm event that will "go-off" at a certain time, it is possible to supply an event handler capable of dealing with a timeout event by communicating an error to the client of the process or by executing exception handling operations capable of maintaining a coherent internal process state. Since only one alarm can be active for each scope, if we want to be able to catch different timeouts, we must create a scope for each invocation of a Web service that we want to monitor. To add a timeout, the designer adds a simple comment that states the duration of the time-frame and the kind of exception handling that the system should use if the time-out expires. For example, the following comment, rendered as XML code

```
<!--  
<timeout>  
    <for>PT5M</for>  
</timeout>  
<exceptionHandling>communicate&terminate  
straits.
```

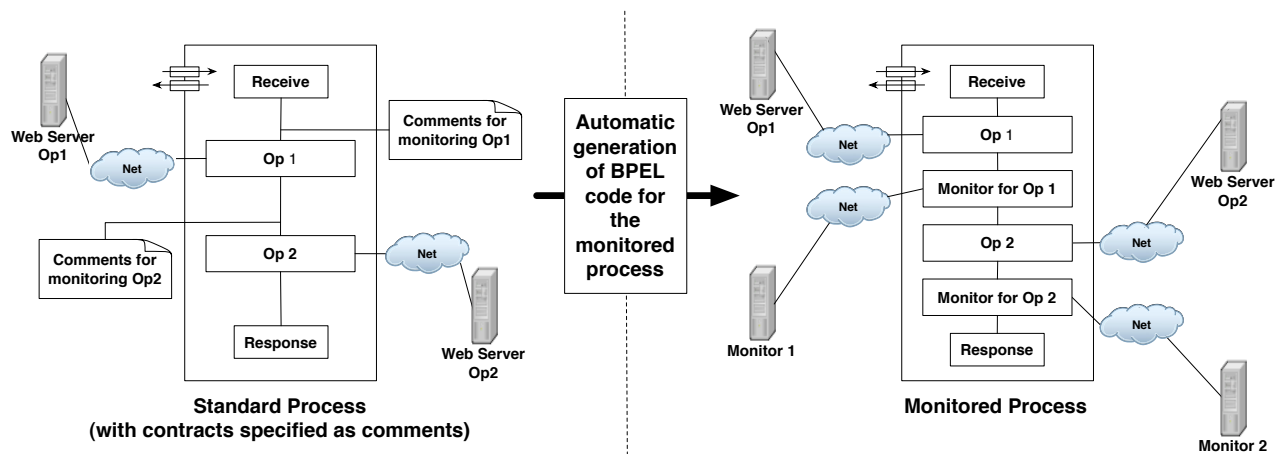


Figure 1: A standard process annotated with contracts can be automatically transformed into a monitored process

```
</exceptionHandling>
-->
```

```
<invoke partnerLink="pLName" portType="pTName"
operation="opName" inputVariable="inVar"
outputVariable="outVar"/>
```

says that the designer specified a time limit of 5 minutes within which the web service invocation must terminate. In case this does not happen, the designer has specified that the process should communicate the error to its client and terminate. The preprocessor transforms such a comment as follows:

```
<scope name = "scopeName">
<eventHandlers>
<onAlarm for="PT5M">
<sequence>
...logic for handling the exception by
communicating the timeout to the client
of the process and terminating...
</sequence>
</onAlarm>
</eventHandler>
<invoke partnerLink="pLName" portType="pTName"
operation="opName" inputVariable="inVar"
outputVariable="outVar"/>
</scope>
```

In this example an `eventHandler`, associated with the scope `scopeName`, will be executed in case the alarm "goes-off", i.e. 5 minutes (PT5M) after entering the scope. Timeouts are of vital importance and should be used in association with service invocations when possible. If a service should stop, the only way of knowing this is, since a stopped service is incapable of communicating any answers or faults to the calling process, to specify a timeout.

2.2 Monitoring external errors

When a process uses external web services to achieve its objectives, it must take into account the possibility of failures that are not under its jurisdiction. We can imagine,

for example, that a web service can suddenly stop working because of an unforeseen internal bug. If the BPEL process ignores this possibility, an error in an external service could cause the failure of the entire process. To cover this situation, we can foresee a solution based on good process design that can be automatically generated if specified through comments. The approach encloses the service invocation in a BPEL scope. By defining a scope for each service invocation, it is possible to define a `faultHandler` that can take care of the failure of the invoked service using a `catchAll` clause. An example of the comments necessary to use this approach could be:

```
<!--
<externalErrors/>
<exceptionHandling>communicate&terminate
</exceptionHandling>
-->
```

```
<invoke partnerLink="pLName" portType="pTName"
operation="opName" inputVariable="inVar"
outputVariable="outVar"/>
```

In this short example, the designer specifies his/her desire to monitor external errors by including the `externalErrors` element in the comment. In case the service invocation cannot be completed correctly, the designer decides that the process should communicate the error and terminate. The monitored version of the process is:

```
<scope name = "scopeName">
<faultHandlers>
<catchAll>
<sequence>
...logic for handling the exception by
communicating the error to the client
of the process and terminating...
</sequence>
</catchAll>
</faultHandlers>
<invoke partnerLink="pLName" portType="pTName"
operation="opName" inputVariable="inVar"
```

```
outputVariable="outVar"/>
</scope>
```

2.3 Monitoring functional contracts

The main concern of our study is to evaluate the use of assertions for the monitoring of functional requirements between web services in an open environment. In contrast to the previous two faults, the monitoring of user requirements imposes a more semantic approach to the issue. What we want to monitor is whether an external web service follows a predefined functional contract.

Although our examples deal with functional constraints, in principle it can be used to monitor compliance with any kind of contract specifying quality of service and other non-functional requirements.

In our solution, a monitor is a web service itself. This choice is one of the key features of our proposals and is motivated by several factors. The first requirement was the use of standard technology to integrate monitoring capabilities: An external service can communicate using BPEL and does not impose any proprietary solution. We also wanted to keep the business logic distinct from monitoring capabilities. The intertwining of the two concepts could negatively impact the business logic of the process and would not allow the designer to change the monitoring capabilities without affecting the process itself. This non-invasive approach permits the adoption of the same interaction paradigm between process and monitors as between process and business services. It is also a way to foster flexibility in terms of monitored features and adopted monitors: The same process can be probed by different monitors and we can change the monitors dynamically with no need for new reconfigurations. Finally, the adoption of external monitoring capabilities does not impact on standard BPEL engines and allows us to execute both monitored and un-monitored processes in the same way. The difference only lies in the pre-processing of the original process. In the first case comments are translated into suitable BPEL code and then transformed processes are executed. In the second case, comments remain unchanged and processes are executed as they are originally. Obviously, a monitor web service can be subject to the same misbehaviors it sets out to probe for (i.e. it can timeout or respond erroneously because of an internal implementation bug). In order to be fail-safe a monitor service can be monitored by a second monitoring service but illustrating this is beyond the goals of this paper.

A monitor needs two pieces of information to be able to complete the requested verification. The first is the definition of the assertions themselves. The second is the data on which we want to verify the assertions. For example, such data can be provided through marshalling of the variables, defined inside the process, on which the assertion predicates. Thus, once a web service has answered to its invocation by providing an appropriate response message, its contents can be serialized into a string or into an XML fragment and sent to the monitor, together with the assertions to be checked. In the next section we will describe the two implementations we have prototyped:

1. The first solution is based upon the characteristics of a flexible environment and a fully fledged object oriented programming language, which supports the possibility to compile new code at run-time and use reflection.

We chose the C# programming language in the .NET environment [14] [17].

2. The second solution is based upon the use of `xLinkIt`, an assertion validating engine[5]. Assertions are expressed in CLIX, a highly expressive XML-based constraint language based upon first order logic. This technology gives us the possibility to express our constraints declaratively. The choice of CLIX was also motivated by its being based upon XML standards, the same standard core to web services and BPEL compositions.

These two implementations follow a common procedure in producing the monitored BPEL process. To monitor user requirements (contracts) within the process both implementations introduce a series of BPEL `assign` activities responsible for preparing all the data necessary to the monitor service (the assertions and the values to be checked), an `invoke` activity responsible for sending this information to the monitor service via a suitable message, a `switch` activity responsible for checking the message returned from the monitor for possible errors and finally, in case of reported errors, a series of activities (consisting of an exception throw and a `faultHandler`) responsible for performing "clean-up" routines or for communicating the error to the client of the process.

3. IMPLEMENTING MONITORS

The two implementations described in this paper support different monitoring capabilities and work at different abstraction levels (implementation vs. specification). The proposed case study will help the reader better understand the differences between the two monitors.

The C# implementation. The first prototype implementation we propose uses .NET technologies. The reason for this choice lies in the flexibility of the .NET framework, which supports the possibility to compile C# code at run-time. By compiling at run-time, we can create new .NET assemblies that can be dynamically loaded and used to provide non anticipated functionalities. Using this technique, we can have the process "create" a personalized monitor at run-time and use it for verifying the defined contracts.

In our prototype implementation we decided to specify the assertions (post-conditions) directly in terms of C# code. These are inserted into the source BPEL process as comments. This code is then converted directly into the .NET monitor assembly capable of verifying all the contracts defined in the process. According to this simplified solution, there is no intermediate step necessary for the parsing of the specified assertions.

In our prototype we use a .NET web service monitor that has a fixed WSDL definition. This way the monitor can be invoked from within any BPEL process, regardless of the number of monitored activities. The WSDL interface publishes two methods that can be invoked by the BPEL process:

- `createMonitorMethods`: This method provides the functionality necessary for the dynamic creation of a .NET assembly containing the monitoring logic for the entire BPEL process. The resulting assembly contains a method for each monitored web service in the BPEL

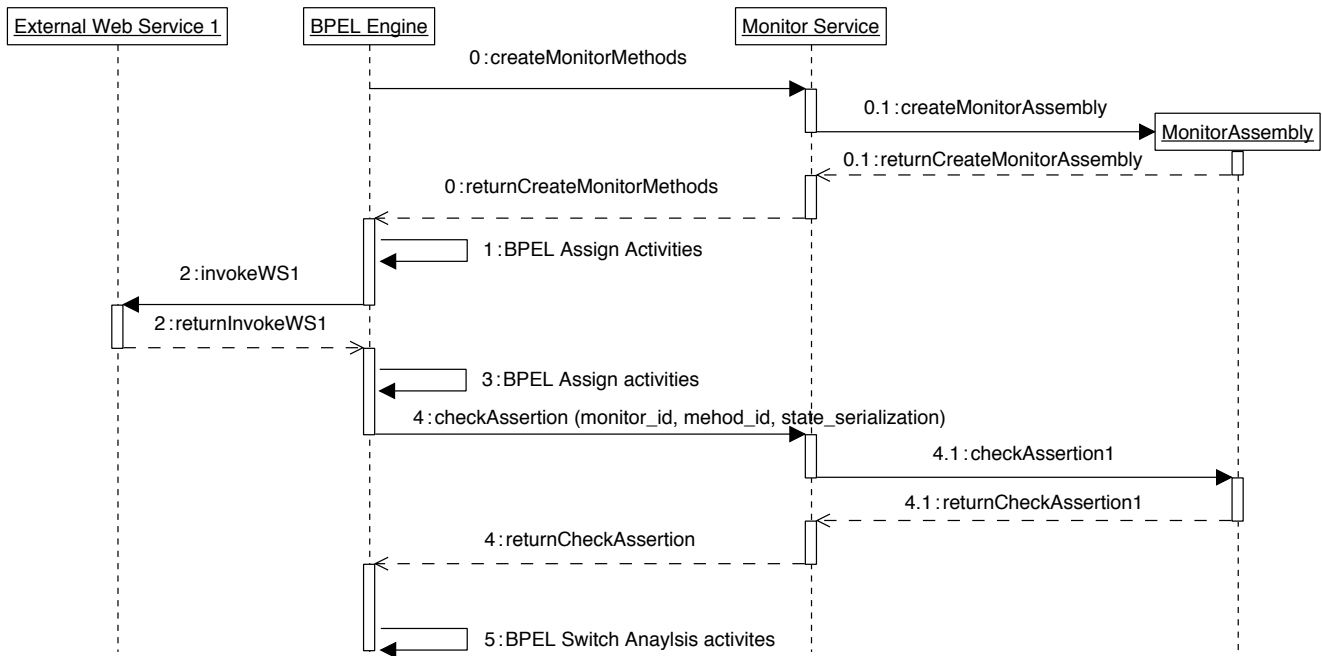


Figure 2: Invocation and monitoring of an external web service in our .NET implementation

process. In our design this web method is typically called, once and for all, at the beginning of the BPEL process. This way it is not necessary to transmit the monitor logic every time we need to monitor some web service invocation.

- **checkAssertion**: This method is a "redirector". Its job is to receive information regarding which web service has been invoked by the BPEL process and, consequently, which assertions need to be verified. Once the necessary information has been recovered, control is passed to the appropriate method of the previously created .NET assembly.

Instead of creating the entire .NET monitor assembly once and for all at the beginning of the monitored BPEL process, it could have been possible to pass only the "step by step" necessary monitoring code as input to the **checkAssertion** method at every invocation. Our choice to send all the necessary C# code at once is justified by implementation reasons. This way only one .NET assembly is created per monitored BPEL process, facilitating the handling of the created assemblies.

This solution, although, raises a number of difficulties that are consequences of the fact that .NET web services are, by default, stateless. A way to solve this problem is to view the monitor web service as a simple gateway to the real monitor, represented by the .NET assembly, which will be dynamically created at the beginning of the BPEL process.

Another problem is that many BPEL processes could be trying to connect to the monitor web service at the same time. The monitor must be able to distinguish between different parties and redirect any requested monitoring activity to the appropriate dynamically created .NET assembly and to the appropriate method contained in that assembly. To

accomplish this, a BPEL process id and a web service id are necessary. By using the process id, the monitor can identify which .NET monitor assembly must handle the incoming verification requests. By using a unique id that identifies each web service within the BPEL process, the monitor service can then identify which method of the .NET monitor assembly to send the request to.

Another difficulty arises when portions of the internal state (i.e. BPEL variable messages) must be sent as input parameters to the monitor web service in order to permit the verification of the appropriate assertions. In our implementation, this problem has been overcome using a simple string serialization of the different variable parts that need to be passed remotely to our external monitor. For this reason, the monitor has been equipped with an internal function capable of deserializing the information correctly.

A normal case of invocation and monitoring of a web service, assuming the .NET monitor has already been created, can be seen as a five step mini-process. A sample instance is described by the sequence diagram of Figure 2, in which the invoked web service is uniquely associated to a method in the .NET assembly monitor by means of a id equal to 1. The necessary steps for a correct invocation of a monitored service are³:

1. Preparation of the SOAP message to be sent to the external web service, whose behavior we wish to monitor. This step can be normally carried out using a series of BPEL **assign** activities.
2. Invocation of the external web service using the message prepared in the previous step.

³Example BPEL code are available at the author's website <http://www.elet.polimi.it/upload/guinea/>.

3. Preparation of the SOAP message to be sent to the external monitor service. This message is prepared using information taken from the response message of the web service invoked in step 2. The information that must be sent to the monitor is the string serialization of a subset of the internal BPEL process state. This step can be carried out using a series of BPEL `assign` activities, capable of providing the necessary serialization capabilities given their support for XPath functions such as `concat()`, `substring()`, etc. Other information must be sent as well: in particular, information regarding the id of the BPEL process and the id of the web service to be checked (necessary to identify the method contained in the .NET monitor assembly dynamically created at the beginning of the BPEL process).
4. Invocation of the monitor web service using the SOAP message prepared in the previous step (the monitor request will be forwarded to the appropriate method in the monitor assembly).
5. Analysis of the response message returned by the monitor service after the invocation in step 4. This analysis can be carried out using the BPEL `switch` activity.

When a monitor response message shows that a contract is violated, different response actions can be undertaken (e.g. perform exception handling activities or terminate orderly by communicating the error to the client of the process). To accomplish this, our implementation exploits some BPEL constructs, such as the possibility to create a `scope` in which to insert these five steps and the possibility to throw exceptions. By throwing a uniquely named exception (done after step 4 if a failure has been noticed) and by catching it with a `faultHandler` associated with the surrounding `scope`, we can separate the code necessary to deal with the failure from the normal sequence code. It is the `faultHandler` that is responsible for invoking the necessary exception handling activities and/or communicating to the client the type of error that has occurred⁴.

The XlinkIt implementation. Our second implementation uses CLIX, a constraint language for the definition of assertions (called "rules") regarding the contents of XML documents. It permits automatic checking of constraints by validating documents against assertions defined in a rule file (or in a set of rule files). This is accomplished by a rule processing engine called `XlinkIt`.

Since CLIX works with XML documents, our BPEL process must prepare them before requesting monitoring activities to the monitor. Precisely, the process must prepare two XML documents every time it requests a monitoring activity: one for the assertions defining the contract and one for the contents upon which to perform the verification of the assertions.

Our implementation of the external monitor consists of a service defined by a very simple WSDL interface. In this interface only one operation has been defined. This web method requests two input strings representing the two XML documents it will have to pass to the `XlinkIt` verification engine. In fact, after receiving the required information from

⁴In both our implementations the default generated behavior is to communicate the error to the client and terminate.

the process, the monitor creates one XML document for the CLIX rules and one for the BPEL process state. These two documents are then passed to the processing engine which produces a third XML document based on the results of its verification activities. This XML document is parsed to see if any faults have been reported and a simple boolean value is returned to the BPEL process, expressing whether the contracts have been verified correctly.

The invocation of a single web service and its monitor are inserted into an appropriately defined BPEL `scope`. This way, specific time-outs, web service errors and contract violations can be treated through appropriately created `eventHandlers` and `faultHandlers`.

A normal case of invocation and monitoring of a web service can be seen as a mini-process that follows the same methodology already described for .NET monitors. The only difference is that no initial invocation is necessary for the creation of a monitor assembly, and that monitoring requests are forwarded to the `XlinkIt` engine instead of to a .NET assembly.

4. A CASE STUDY

To demonstrate the validity of our approach, we present a case study in the domain of IT certification.

In this example, an IT firm called *ServiceSoft* decides to promote its certification program by giving college students, enrolled in a software engineering course, the chance to try a certification test for free. The idea is to provide the students with the necessary training material (mainly books) and to have them take the test directly at their university, within the already existing educational structure. To do this, *ServiceSoft* decides to sign an agreement with the famous publisher *IT Books*, in order to have them provide the necessary study material.

To manage the necessary organizational activities, which must take into account the coordination of three entities (the university, the publisher and itself), *ServiceSoft* decides to define a process that uses services previously made available by the three entities. The process is defined using BPEL and executed on a Collaxa BPEL Server [6].

The services used in the composition are:

- **Services provided by the university:** After defining a policy for the organization of events involving external parties, the university decided to provide services useful for gathering information as to which classrooms can be used for the event and for assigning enrolled students to the classrooms. The web service can also provide information about students (i.e. name, surname, address for didactical and promotional material, etc.). *ServiceSoft* is looking forward to using these services to acquire information regarding the students in order to: enroll them correctly, organize the logistics of the event and have *IT Books* send them the study material once the subscription process is completed.
- **Services provided by *IT Books*:** The company created a web service for business interactions. Through this web service, it is possible to order books and have them sent to a certain address. *ServiceSoft* is looking forward to using this service for buying and sending the study material to the enrolled students.

- **Services provided by *ServiceSoft*:** These services were created to facilitate the organization of free IT certification trial offers. Through these services, the company can check for vacant trial offers and assign one to a specific person by storing his/her participation details in their data storage system. From this point on we refer to these services as the "organizational" services.

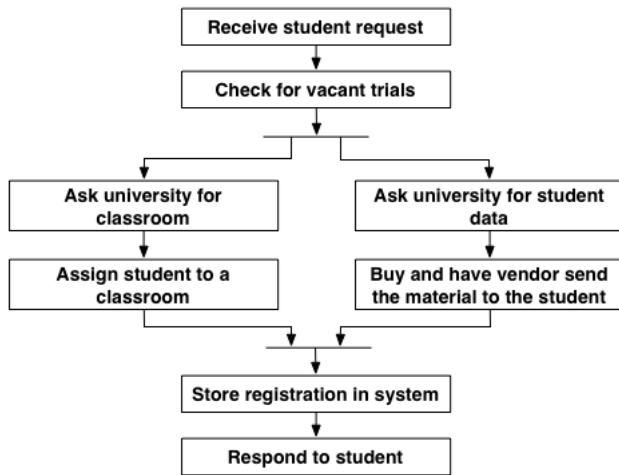


Figure 3: The unmonitored process

ServiceSoft decides to use a monitored process in order to be able to react to process faults as fast as possible.

4.1 Designing the unmonitored process.

We start the design by sketching an unmonitored process, responsible for the correct enrollment of a single student to the trial offer. The defined process, shown in Figure 3 as an activity diagram, can be seen as a 8 step procedure:

1. The first step of the process consists of receiving the enrollment request from the student. The required information is the student's university ID number.
2. Upon receiving a valid ID number, the process asks the organizational services if trial offers are available or if the maximum number of students that can be enrolled has already been reached.
3. The process asks the university services in which classroom the enrolling student will have to go to undertake the trial.
4. The process assigns the student to a specific classroom.
5. The process asks the university for more information regarding the student: in particular, information necessary for sending him/her the training material.
6. The process uses the service offered by *IT Books* to buy the training material and, using the data acquired in step 5, to send the material to the student.
7. The process stores the enrollment in its data storage facilities using the organizational services.

8. The process communicates the successful enrollment to the student.

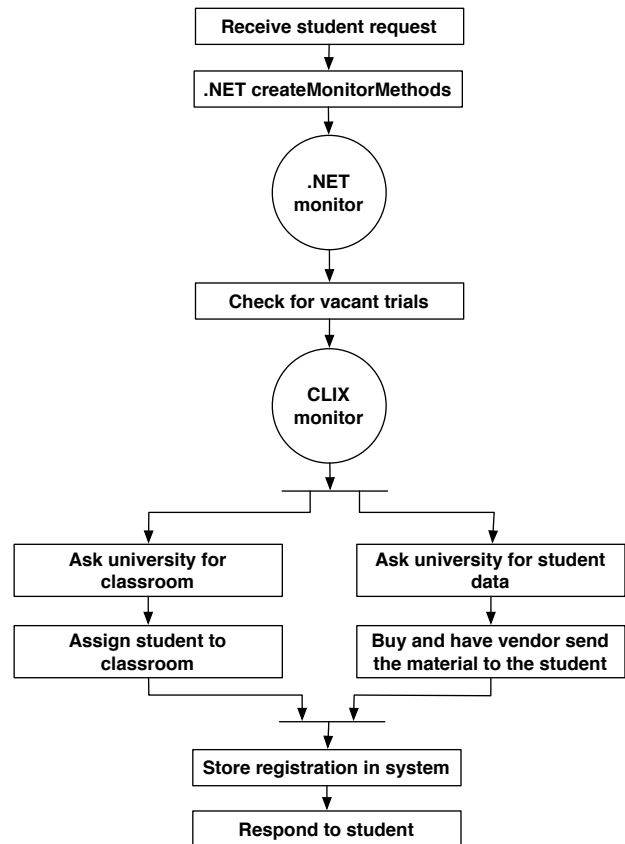


Figure 4: The monitored process

The web service defined by the BPEL process is initiated by receiving a request made by a student to a specific WSDL port. A student's desire to enroll in the trial is represented by a service method call made to the process. This method call consists in passing his/her university ID number as a parameter.

Before illustrating the monitored version of the process, notice that the 8 steps are not arranged in a pure sequence, but the sequence of steps 3 and 4 is executed in parallel to the sequence of steps 5 and 6. In fact, step 7 can be undertaken only after the parallel activities have completed without errors.

After sketching the process, let us concentrate on the contracts that we want to check:

1. The ID passed to the process by the student must appear in the set of IDs associated with students entitled to the free trial offer.
2. Vacant trials must be signaled within 5 minutes.
3. The number of available vacant trials acquired through the `Receive student request` operation must be greater than zero and less than the maximum number of free trials originally offered by *ServiceSoft*.

Because of their different characteristics, the first contract exploits the .NET monitor, the second a good process design and the third the XlinkIt monitor.

4.2 Designing the monitored process

The following description of the monitored process illustrates the results of the automatic generation of BPEL code, from the comments inserted by the designer, and some results about its execution.

Integrating the .NET monitor. Since not every student in the university is given the possibility to enroll in this trial, but only those participating in the Software Engineering course, the first step of the process must be to verify the inserted ID against those associated with students entitled to the free trial. For this reason, the university provided, a file containing the IDs of the eligible students. Instead of coding the verification into the BPEL process or coding an appropriate web service responsible for checking if the ID is contained in the set of eligible students, in this example we decided to specify a constraint to the operation **communicating the ID to the process**. This operation is the activity that kicks-off the entire BPEL process; it is an external operation that is undertaken by the student and an operation we must monitor. In this case we decided to use a .NET monitor and to specify a post-condition on the value of the inserted ID in terms of C# code, provided as comments inserted in the process (and not reported here for space reasons).

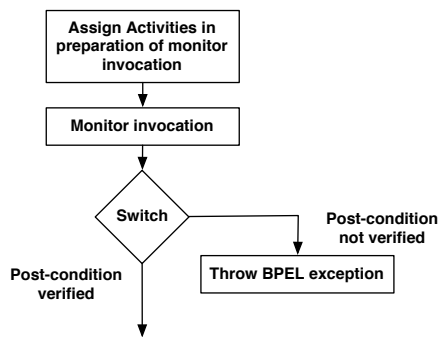


Figure 5: BPEL code added for each monitored invocation

As we explained in Section 3, to use a .NET monitor it is necessary to invoke the .NET service to dynamically create the needed .NET monitor assembly. This is done by invoking the method `createMonitorMethods` of the .NET monitor service. After completing this step, a scope is created for the `receive` activity and the monitoring activities which are to be inserted after it. The monitoring activities (described by circles in Figure 4) are defined (see Figure 5) in terms of: (1) BPEL `assign` activities for preparing the monitor invocation, (2) an invocation of the monitor, (3) a BPEL `switch` activity responsible for interpreting the monitor response, and (4) a series of activities responsible for exception handling.

In Figure 6 we can see what happens when a non entitled student requests a free trial by passing his/her university ID

```

=> client (onResult)

[2004/05/26 14:58:59]
Skipped callback "onResult" on partner "client".

<output>
<part name = "payload">
<univExampleResponse xmlns="http://acm.org/samples">
<result>Student ID inserted not valid.</result>
</univExampleResponse>
</part>
</output>
  
```

Figure 6: The monitor communicates a violated contract

to the process. The .NET monitor discovers that the student is not eligible and causes the BPEL process to throw an internal exception, which is caught by an appropriately defined `faultHandler` which terminates the process. It also sends a message to the client, in this case the standard interface provided by Collaxa, saying "Student ID inserted not valid."

Integrating the XlinkIt monitor. The other activity we decided to monitor is the invocation of the service responsible for informing the process if there are vacant trial offers available. As we mentioned in Section 4.1, two things are to be monitored: (1) the service invocation must be completed in less than 5 minutes and (2) it must respond with a value that is functionally correct (it must be greater than zero and less than the maximum number of free trials originally offered). Both are monitored using the XlinkIt monitor. This monitor gives us the possibility to easily state assertions declaratively. The nature of the monitoring activity in this case is clearly different from the previous case. The comments inserted in the process to automatically generate the monitoring activity are:

```

<!--
<contents>
  <var>
    <name>var1</name>
    <origin>varInternalToBPELSpec</origin>
    <path>/trial/availability</path>
  </var>
</contents>
<assertion>
  <and>
    <greater>
      <var>var1</var>
      <value>0</value>
    </greater>
    <less>
      <var>var1</var>
      <value>max_num_trials</value>
    </less>
  </and>
</assertion>
<timeout>
  <for>PT5M</for>
</timeout>
<exceptionHandling>communicate&terminate
  
```

```
</exceptionHandling>
-->
```

By specifying "PT5M" in the element named `timeout`, the designer indicates he wants the process to wait for the correct conclusion of the service invocation for no more than 5 minutes. This part of the comment will produce an `event-Handler` responsible for catching the event of the alarm going-off.

The functional contract is specified by two elements. The first, named `contents`, contains information on how to obtain the variable values we want to send to the monitor. It is, in turn, made up of three sub-elements: one containing the name of the source BPEL variable from which to extract the value to send to the monitor, one containing an XPATH expression that defines how this extraction must be accomplished, and one containing the variable name that will be used to reference this data. In this case, the designer associates `var1` to the value contained in the `varName` variable and indicated by the `/trial/availability` expression. The second element, named `assertion`, contains the definition of the rules to be used to check the value contained in `var1`. The designer has specified that the value contained in `var1` must be greater than zero and less than the maximum number of trials originally offered by *ServiceSoft*.

The BPEL code produced for the monitoring of the functional contract is similar to that prepared for the .NET monitor. A BPEL `scope` is created to surround the external web service invocation and the BPEL code necessary for its monitoring: a BPEL `assign` activity is responsible for preparing the input message for the monitor service, a BPEL `invoke` activity for invoking the monitor and a BPEL `switch` activity for interpreting the monitor response and launching the internal BPEL exception.

Finally, in our comments, the designer indicated what kind of exception handling activities are to be undertaken if needed. The chosen behavior (both for the time-out event and the contract violation) is to communicate the fault to the client of the process and terminate. This behavior is defined in the element named `exceptionHandling`.

In our example, the fragment of XML automatically created that specifies the contents on which to verify the assertions is:

```
<trial>
<availability>num_vacant_trials</availability>
</trial>
```

The contents have an XML structure that is automatically generated from the specifications provided in the element named `contents` in the comments inserted by the designer. Precisely, information contained in the sub-element named `path` is used. The fragment of XML automatically created that specifies the assertions is:

```
<clix:and>
<clix:greater op1="number($var1)" op2="0"/>
<clix:less op1="number($var1)"
  op2="max_num_trials"/>
</clix:and>
```

The assertion states, as expected, that the variable "var1" must contain a value greater than "0" and less than the maximum number of trials that *ServiceSoft* has provided for this event. If this is not true the situation is clearly erroneous and must be dealt with.

5. RELATED WORK

The approach presented in this paper clearly originates from the many assertion systems available for programming languages. Besides the already cited Anna and APP, we can also mention Eiffel [15], as the well-known programming language that embeds assertions, and iContract [16] as one of the best assertion systems for Java. Anna [8], for example, is a language extension to Ada that allows the definition of intended behaviors for Ada programs. It is based on first order logic and, as in our work, its constructs are inserted as formal comments in the source code of the program. A suitable preprocessor transforms these annotations into checking functions that can control potential violations in the program[9][1].

These systems only provided the initial idea, but nowadays there are other proposal that address the problem of monitoring the composition of services (components). For example, we can mention BPELJ[2] as a general approach to the problem of composing web services that can also be used to solve the monitoring problem. BPELJ embeds pieces of Java code into BPEL processes. This Java code allows the designer to implement probes by programming in the small, in contrast to programming in the large, typical of BPEL solutions. This approach can be used to accomplish the monitoring of contracts in a way similar to our .NET monitor. The difference is that our approach lies completely within the BPEL standard and does not require any extension to execution engines.

JOpera[10] is another system that support integration of components (and thus also compositions of web services). JOpera provides a visual composition definition tool (which produces a compiled Java program) and a monitoring environment. The monitoring environment permits the designer to view the state of a process and the contents of its input and output parameters. The monitoring is, in fact, conceived differently from our approach. It is not possible to define functional contracts to "monitor". Our approach is definitely compatible with JOpera and the idea of using external monitors can be very easily integrated into JOpera processes. In this way it is possible to take advantage of the benefits of a well defined process support system, while introducing the possibility to define contract-oriented processes.

Other approaches to the monitoring problem can be found outside of the service composition scenario. An efficient decentralized monitoring algorithm and implementation is proposed in DIANA[13], a distributed systems application development framework. This algorithm is tailored towards the monitoring of the execution of a distributed program, in order to check for violations of safety properties. It is based on formulae written in PT-DTL (Past Time Distributed Temporal Logic). This logic permits the definition of formulae that are aware of the global state of a distributed program. The monitoring logic is distributed throughout distributed local monitors that evaluate remote expressions by using a KnowledgeVector that keeps track of the last known values of remote expressions and formulae.

Finally, we mention the work by Robinson [18], who uses run-time monitors to match requirements against system executions. His work concentrates on high-level requirements. From them, he discovers obstacles, and derives monitors. In contrast, we are closer to the implementation and concentrate on violations of interface contracts in dynamic service

compositions, leaving the designer more freedom in tailoring the degree of accuracy he wants to adopt to define pre and post conditions.

6. CONCLUSIONS AND FUTURE WORK

The paper presents our approach for monitoring the execution of composed Web services. The approach starts from BPEL processes: The user annotates the process and the system transforms it into a monitored process. The processor transforms annotated processes into other BPEL processes that exploit exception handling, to address timeouts and errors in called services, and invocations of an external monitor service, to assess the correctness of service compositions.

The paper exemplifies the idea of external monitors by means of two distinct implementations. The two solutions present different characteristics, which are exemplified and discussed by means of a case study. If we think of the first solution, assertions are fragments of C# code that return a boolean value. This means that the designer can add special-purpose code just for monitoring/probing the process. This also permits the interaction with other components, like data sources or suitable graphical interfaces. The result is that the information necessary to the monitoring activities can be obtained by the monitor itself. Necessary data do not need to be a part of the internal state of the BPEL process. This is possible because the C# code is written knowing exactly what components and capabilities are available on the monitor service's site. This, although, makes for a lower level solution where assertions can be written to exploit these foreknown capabilities. In contrast, the second solution does not provide the designer with the same freedom as the first proposal, but supports a pure model-oriented approach by supplying the typical constructs of assertion languages.

Our future work will be devoted to further validate the approach. We think that our external monitors are general enough to address both functional and non-functional requirements: functional requirements have been addressed by this paper, while the definition of contracts based on QoS parameters needs to be further investigated. In this paper, we did not concentrate on performance or usability issues. We are, although, aware that many optimizations are possible. For example, a better separation between business logic and annotations could facilitate the deployment within enterprises, where it could be imperative to distinguish between different monitoring assertions for different stake-holders. In our future work, we will also investigate the possibility of defining more complex exception handling routines. For example, we are working on a library of pre-defined actions that designers can add to their processes by means of suitable annotations.

7. REFERENCES

- [1] A. Hahmood and E.J. McCluskey. Concurrent Error Detection Using Watchdog Processors - A Survey. Technical report, 1985.
- [2] BEA and IBM. BPELJ: BPEL for Java. 2004. <http://www-106.ibm.com/developerworks/webservices/library/ws-bpelj/>.
- [3] BEA and IBM and Microsoft and SAP. Web Service Policy Framework (WSPolicy). 2003.
- [4] BEA and IBM and Microsoft and SAP and Siebel. Business Process Execution Language for Web Services Version 1.1. 2003.
- [5] C. Nentwich and L. Capra and W. Emmerich and A. Finkelstein. XlinkIt: a Consistency Checking and Smart Link Generation Service. *ACM Transactions on Software Engineering and Methodology*, pages 151–185, May 2002.
- [6] Collaxa. Collaxa: Model, deploy and Manage BPEL Business Processes. 2004. <http://www.collaxa.com>.
- [7] David S. Rosenblum. A Practical Approach to Programming with Assertions. *IEEE Transactions on Software Engineering*, 21(1), Jan 1995.
- [8] D.C. Luckham. Programming with Specifications: An Introduction to Anna, A Language for Specifying Ada Programs. *Texts and Monographs in Computer Science*, Oct 1990.
- [9] D.S. Rosenblum and S. Sankar and D.C. Luckham. Concurrent Runtime Checking of Annotated Ada Programs. In *Proceedings of the 6th Conf. Foundations Software Technology and Theoretical Computer Science*, pages 179–188, 1986.
- [10] ETH Zurich Department of Computer Science Institute for Pervasive Computing. Jopera: Process Support for Web Services, 2004. <http://www.iks.inf.ethz.ch/jopera/>.
- [11] IBM alphaWorks. IBM Business Process Execution Language for Web Services Java Run Time. 2004. <http://www.alphaworks.ibm.com/tech/bpws4j>.
- [12] James Skene and D. Davide Lamanna and Wolfgang Emmerich. Precise Service Level Agreements. In *Proceedings of the 26th International Conference on Software Engineering (ICSE2004)*, pages 179–188, 2004.
- [13] Koushik Sen and Abhay Vardhan and Gul Agha and Grigore Rosu. Efficient Decentralized Monitoring of Safety in Distributed Systems. In *Proceedings of the 26th International Conference on Software Engineering (ICSE2004)*, pages 418–427, 2004.
- [14] Jesse Liberty. *Programming C#, Third Edition*. O'Reilly, 2003.
- [15] B. Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, N.Y., second edition, 1997.
- [16] Reto Kramer. iContract - The Java Design by Contract Tool. In *Technology of Object-Oriented Languages. TOOLS 26 Proceedings*, pages 295–307, Aug 1998.
- [17] Jefferey Richter. *Applied Microsoft .NET Framework Programming*. Microsoft Press, 2002.
- [18] W. Robinson. Monitoring web service requirements. In *Proceedings of the International Conference on Requirements Engineering*, 2003.
- [19] The World Wide Web Consortium (W3C). Web Service Choreography Interface (WSCI) 1.0. 2002.
- [20] V. Tasic and K. Patel and B. Pagurek. WSOL - Web Service Offerings Language. In *Proceedings of the Workshop on Web Services, e-Business, and the Semantic Web - WES (at CAiSE02)*, volume 2512 of *Lecture Notes in Computer Science*, pages 57–67. Springer-Verlag, 2002.