Towards Dynamic Monitoring of WS-BPEL Processes

Luciano Baresi and Sam Guinea

Dipartimento di Elettronica e Informazione - Politecnico di Milano,
Piazza L. da Vinci 32, I-20133 Milano, Italy
{baresi, guinea}@elet.polimi.it

Abstract. The intrinsic flexibility and dynamism of service-centric applications preclude their pre-release validation and demand for suitable probes to monitor their behavior at run-time. Probes must be suitably activated and deactivated according to the context in which the application is executed, but also according to the confidence we get on its quality. The paper supports the idea that significant data may come from very different sources and probes must be able to accommodate all of them.

The paper presents: (1) an approach to specify monitoring directives, called monitoring rules, and weave them dynamically into the process they belong to; (2) a proxy-based solution to support the dynamic selection and execution of monitoring rules at run-time; (3) a user-oriented language to integrate data acquisition and analysis into monitoring rules.

1 Introduction

The flexibility and dynamism of *service-centric* applications impose a shift in the validation process. Conventional applications are thoroughly validated before deployment, and testing is the usual means to discover failures before release. In contrast, service-centric applications can heavily change at run-time: for example, they can bind to different services according to the context in which are executed or providers can modify the internals of their services. New versions of selected services, new services supplied by different providers, and different execution contexts might hamper the correctness and quality levels of these applications. Testing activities cannot foresee all these changes, and they cannot be as powerful as with other applications: we need to shift validation to run-time, and introduce the idea of *continuous monitoring*.

Runtime monitors [6] are the "standard" solution for assessing the quality of running applications. Suitable probes can control functional correctness, and also the satisfaction of QoS parameters, but web services introduce some peculiar aspects. Functional correctness can be easily monitored by analyzing the data exchanged among services, but service-centric applications also require that the many QoS aspects be monitored with data that can be collected at different abstraction levels. We can analyze the SOAP messages exchanged between client and provider, trace the events generated during execution, and collect data from external metering tools. All these options must be accommodated in a general framework that lets designers choose the values of interest and the way they want to collect them.

B. Benatallah, F. Casati, and P. Traverso (Eds.): ICSOC 2005, LNCS 3826, pp. 269–282, 2005.

Current technology for executing (composed) services, like the WS-BPEL engines available in these days, does not support monitoring. It only allows designers to intertwine the business logic with special-purpose controls at application level, thus hampering the separation between the definition of the application (i.e., the WS-BPEL process) and the way it can be monitored. Designers must be free to change monitors without affecting the application, and the actual degree of control must be set at run-time. In fact, since monitoring impacts performance, the user must be able to change the amount of monitoring while the application executes to adjust the ratio between control and performance.

In this context, the paper presents an approach towards the *dynamic* monitoring of WS-BPEL processes. It proposes external *monitoring rules* as means to dynamically control the execution of WS-BPEL processes. This separation allows different sets of rules to be associated with the same process. Monitoring rules abstract Web services into suitable UML classes, and use this abstraction to specify constraints on execution. Assertions are specified in WS-CoL (Web Service Constraint Language), a special-purpose assertion specification language that borrows its roots from JML (Java Modeling Language [11]), and extends it with constructs to gather data from external sources (i.e., to interact with external data collectors).

Besides constraining the execution, monitoring rules provide parameters to govern the degree of run-time checking. After weaving selected rules into the process at deployment time, the user can set the amount of monitoring at run-time by means of these parameters (see Sections 3 and 4). The weaving introduces a proxy service, called *monitoring manager*, which is responsible for understanding whether a monitoring rule must be evaluated, interacting with the external services, and calling known data analyzers (monitors) to evaluate specified constraints. This solution can be seen as a feasibility study (proof of concept) before embedding the manager in a WS-BPEL engine and letting monitoring rules become part of the execution framework.

The approach is demonstrated on a simple example taken from [8]. Even if the proposal is suitable for checking both functional and non-functional constraints, here we only address QoS related monitoring rules since functional aspects were already studied in [7].

This paper is the natural continuation of the work already presented in [7], and its novel aspects are: (1) the idea of monitoring rules, (2) WS-CoL to specify constraints on execution, (3) the capability of setting the degree of monitoring at run-time, and (4) the proxy-based solution to enact the monitoring rules.

The rest of the paper is organized as follows. Section 2 introduces the monitoring approach, while Section 3 describes monitoring rules and Section 4 introduces the monitoring manager. Section 5 surveys similar proposals and Section 6 concludes the paper.

2 Monitoring Approach

The ideas behind the monitoring approach presented in this paper come from assertion languages, like Anna (Annotated Ada [4]) and JML (Java Modeling Language [11]), which let the user set constraints on program execution by means of suitable comments added to the source code. Similarly, we propose monitoring rules to annotate WS-BPEL

processes and constrain their executions both in terms of functional correctness and satisfiability of the QoS agreements set between the client, which runs the WS-BPEL specification, and the providers, which supply the services invoked by the WS-BPEL process.

Monitoring rules are blended with the WS-BPEL process at deployment-time. The explicit and external definition of monitoring rules allows us to keep a good separation between business and control logics, where the former is the WS-BPEL process that implements the business process, and the latter is the set of monitoring rules defined to probe and control the execution. These rules also comprise meta-level parameters that allow for run-time tailoring of the degree of monitoring activities. This separation of concerns lets designers produce WS-BPEL specifications that only address the problem they have to solve, without intertwining the solution and the way it has to be checked. Different monitoring rules (and/or monitoring parameters) can be associated with the same WS-BPEL process, thus allowing the designer to tailor the degree of control to the specific execution context without any need for reworking the business process. Moreover, a good separation of concerns allows for a neater management of monitoring rules, and it is an effective way to find the right balance between monitoring and performance.

Besides separation of concerns, the approach was conceived with the goal of reusing existing technology to ease the acceptability of the approach and foster the adoption of monitoring techniques.

All these reasons led to the monitoring approach summarized in Figure 1. It starts as soon as a WS-BPEL process exists (or the designer starts working on it):

- Monitoring rules are always conceived either in parallel with the business process or just after designing it. These rules are associated with specific elements (for example, invocations of external services) of the business process, and are stored in monitoring definition files.
- When the designer selects the rules to use with a specific execution, $BPEL^2$ instruments the original WS-BPEL specification to make it call the monitoring manager.

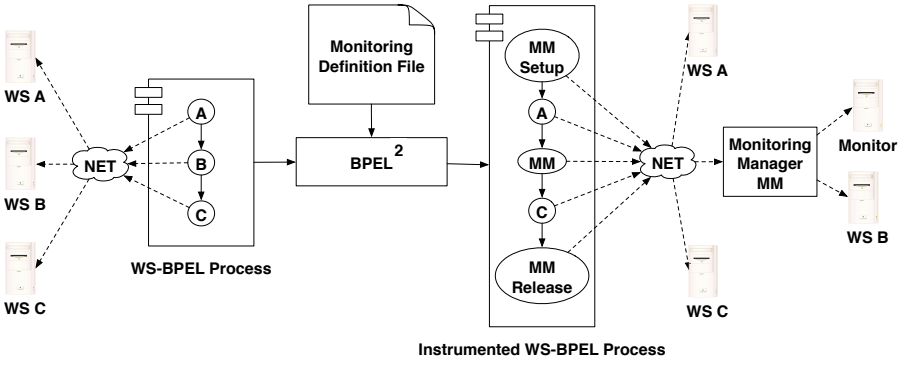

Fig. 1. Our Monitoring Approach

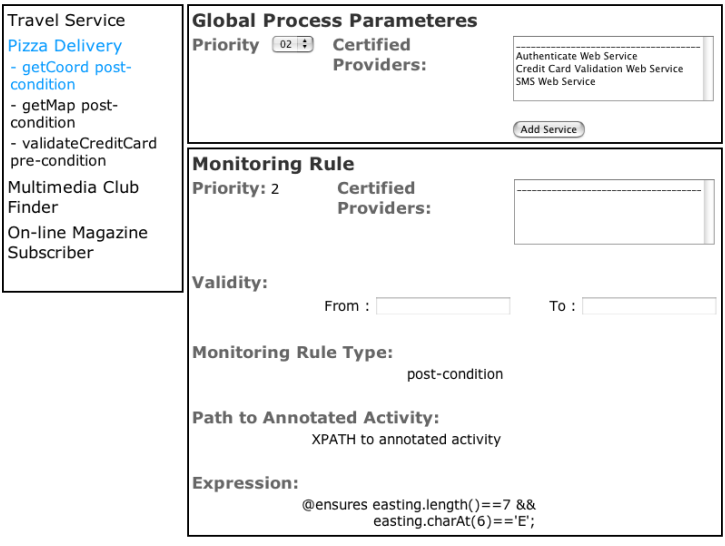

Fig. 2. The monitoring manager's interface

- When the instrumented WS-BPEL process starts its execution, it calls the monitoring manager whenever a monitoring rule has to be considered. The actual invocation of the monitor, that is, the actual analysis of execution/QoS data depends on the current status of the manager. For example, if a rule has priority lower than the current one, the manager skips its execution and calls the actual service directly.
- The designer has a special-purpose user interface (see Figure 2) to interact with the monitoring manager and change its status. This happens when the designer wants to change the impact of monitoring at run-time without re-deploying the whole process.
- If some constraints are not met, that is, if some monitoring rules are not satisfied, the monitoring manager is in charge of letting the WS-BPEL process know. It could also activate *recovery actions* specified in the monitoring rules, but this topic is not part of this paper, and recovery actions are still work in progress.

2.1 Weaving

Code weaving is performed by the BPEL^2 pre-processor. Its job is to parse the monitoring rules associated with a particular process and to add specific WS-BPEL activities to the process in order to achieve dynamic monitoring . If the rule embeds a post-condition to the invocation of an external web service, BPEL^2 substitutes the WS-BPEL invoke activity with a call to the monitoring manager (Figure 3), preceded by WS-BPEL assign activities that prepare the data that have to be sent to the monitoring manager, and followed by a switch activity which checks the monitoring manager's response. The monitoring manager is then responsible for invoking the web service that is being monitored and for checking its post-condition with the help of an external data analyzer.

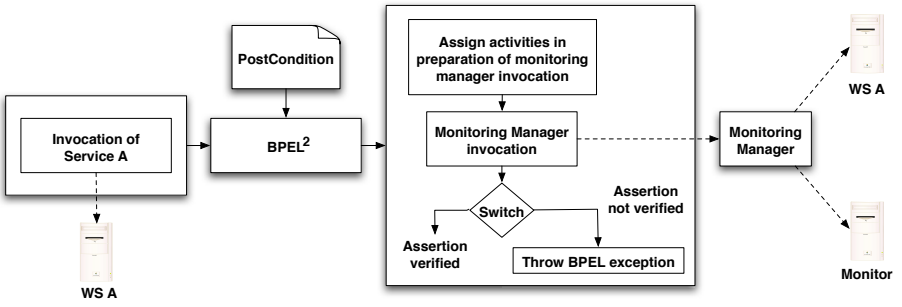

Fig. 3. The effects of weaving

Depending on the response it receives from the monitoring manager, the process flow can either continue or stop (see Figure 3). Pre-conditions are treated the same way, except that the monitoring manager first checks the pre-condition, and only if it is verified correctly does it then proceed to invoke the web service being monitored.

If the rule represents an invariant on a scope, BPEL^2 translates it as a post-condition associated with each of the WS-BPEL activities defined in the scope. If the rule is a punctual assertion then a single call to the monitoring manager is added, together with the corresponding WS-BPEL assign and switch activities.

BPEL^2 always adds to the WS-BPEL process an initial call to the monitoring manager to send the initial configuration such as the monitoring rules and information about the services it will have to collaborate with (see `MM Setup` in Figure 1). BPEL^2 also adds a ”release” call to the monitoring manger to communicate it has finished executing the business logic (see `MM Release` in Figure 1). This permits the monitoring manager to discard any configurations it will not be needing anymore. Every call to the monitor manager (which is not a setup or a release call) is also signed with a unique incremental identifier. This is used for matching the manager call to the specific rules and the data stored in the monitoring manager during setup.

This solution does not require any particular tool to run and monitor WS-BPEL processes. Once the weaving of rules has been performed, the resulting process continues to be a standard WS-BPEL process which simply calls an external proxy service to selectively apply specified monitoring rules.

3 Monitoring Rules

Monitoring rules reflect the ”personal” monitoring needs that single users of WS-BPEL processes may have. Every time a WS-BPEL process is run, different monitoring activities should be enacted, depending on ”who” has invoked the process. This requires the ability to define and associate monitoring activities to a single WS-BPEL process instantiation, or execution. These definitions are conceived by producing a monitoring definition file.

The monitoring definition file follows the structure illustrated in Figure 4. The information it provides is organized into three main categories: *General Information*, *Initial*

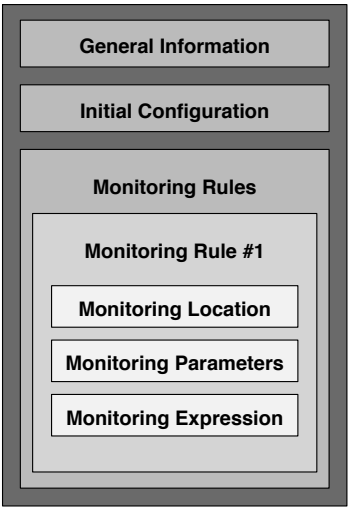

Fig. 4. Monitoring Definition

Configuration, and *Monitoring Rules*. The first part provides generic data regarding the WS-BPEL process to which the monitoring rules will be attached. The second part contains values that are associated with the process execution as a whole and can impact the amount of monitoring activities that will be performed at run-time. This concept will be further analyzed in Section 4. The third part, the monitoring rules, represent the core of the monitoring definition. They are organized in *Monitoring Location*, *Monitoring Parameters*, and *Monitoring Expressions*.

The first element indicates the exact location in the WS-BPEL process in which the monitoring rule must be evaluated. The second element contains a set of monitoring parameters, meta level information that define the context of the monitoring rule itself. These parameters influence the actual evaluation of the rule, and can even impede its run-time checking. Since we envisage the existence of multiple external monitors, the type of monitor that should be used for the given rule is an important parameter. Besides this, we currently consider three parameters (but many others could easily be added in the future[1]). The three parameters considered so far are:

Priority. It is a number between one and five indicating the level of importance that is associated with the rule. A priority level of one indicates a very low priority level, while a priority level of 5 indicates a very high priority level. The idea is that a process can run at various levels of priority. Given a process priority, any monitoring rule with a priority level inferior to this threshold would not be considered at run-time. This makes it possible to execute the same business logic with different degrees of monitoring.

[1] The context could be more complex and address the physical location in which the process is executed, or interact with the device on which the process executes through interfaces such as WMI (Windows Management Instrumentation).

Validity. The user defining the monitoring rules can decide to associate a time-frame with a monitoring rule. Every time a process execution occurs within this time-frame, the monitoring rule is checked; while, should it occur outside the time-frame, it would be ignored. This can be useful when a service invocation must be initially monitored for a certain amount of time before deciding that it can be trusted.

Certified Providers. It is a list of providers that gives us a way of indicating that the monitoring activity does not have to be executed if the actual service is supplied by one of the providers in the list. This is because we envisage monitoring playing a key role in systems living in highly dynamic environments, and for this reason we imagine that a specific service with which to do business could be chosen dynamically. We are never entirely sure of "who" will really be providing that service at run-time. In fact, even when a service has been chosen statically, it can still need to be substituted at run-time in the wake of erroneous situations.

The third and last element, the monitoring expression, states the constraint that has to be evaluated.

The monitoring definition file is mainly a container for the definition of the monitoring rules that are to be executed at run-time and of the conditions at which they can be ignored. Obviously, this leads to the need of specific languages for identifying the locations and for defining the expressions embedded in the rules.

3.1 Locations

In our approach we want to monitor pre- and post-conditions associated with the invocations of external web services, invariants that can be attached to WS-BPEL scopes, and punctual assertions indicating a property that must hold at a precise point of execution. While defining locations, we specify two things: the kind of condition we want to monitor, and in which point of the process definition we want to monitor it. For the first part, we use a keyword indicating whether the monitoring rule specifies a *pre-condition*, a *post-condition*, an *invariant*, or an *assertion*. For the second part, we use an XPATH query capable of pointing out where the rule has to be checked in the process, independently of the fact that the run-time checking could later be dynamically switched off. In the first two cases (pre- and post-conditions) the XPATH query indicates the WS-BPEL invoke activity to which we associate the rule, in the case of an invariant it indicates the WS-BPEL scope to which we associate it, and in the case of an assertion it indicates any point of the WS-BPEL process (in this case we indicate the WS-BPEL activity prior to which the assertion must hold). Regarding pre- and post-conditions, we are only interested in attaching monitoring rules to WS-BPEL activities that can in some way modify the contents of the process' internal variables. We are not interested in attaching monitoring rules to activities that are used by WS-BPEL to define the process topology. Therefore, we assume that pre- and post-conditions can be attached to WS-BPEL invoke activities, post-conditions to receive activities, and pre-conditions to reply activities. We also assume that post-conditions can be associated with *onMessage* branches in WS-BPEL pick activities. The reason for this is that although pick activities contribute to the process topology, they also help define the internal state of the process, and therefore should be monitored.

For example, recalling the *Futuristic Pizza Delivery* example presented in [8], if we want to define a post-condition on the invocation of the operation named `getMap` published by the `MapWS` web service and linked to the WS-BPEL process through partnerlink `MapServicePartnerLink`, we would define the location as[2]:

```
type = "post-condition"
path = "//:invoke[@partnerLink="lns:MapServicePartnerLink" and
                    @operation="getMap"]"
```

3.2 Expressions

For monitoring expressions, we propose to reason on an abstraction of the WSDL definitions of the services the WS-BPEL process does business with. Depending on the degree of dynamism, these could be the actual services used by the application, or abstract descriptions of the services the process would like to bind to (dynamic binding is not treated in this paper). To do this we use a tool based on Apache AXIS WSDL2Java [2]. The tool permits us to reason on stereotyped class diagrams that represent the classes that are automatically extracted from a WSDL service description. In the tool, a web service becomes a ≪service≫ class that provides one public method for each service operation and no public attributes. Similarly, for each message type defined in the WSDL a ≪dataType≫ class is introduced, containing only public attributes and no methods. Figure 5 shows a `MapWS` ≪service≫ class that provides a single method called `getImage`. The exposed method takes a `GetImageRequest` ≪dataType≫ as input and produces a `GetImageResponse` ≪dataType≫ as output. This way we can state our pre- and post-conditions by referring to these classes. If we want to express an invariant, we can only express conditions on variables visible within the WS-BPEL scope to which the invariant is attached. Since internal WS-BPEL variables are structured as simple or complex XSD types, the automatic translation to stereotyped class diagrams can still be achieved. The same holds for expressions that are punctual assertions. The only difference lies in the visibility of the variables the expression can refer to.

Expressions are defined using WS-CoL , inspired by the light-weight version of JML (Java Modeling Language [11]). WS-CoL further simplifies it and introduces a set of instructions for specifying how we can retrieve data that are external to the process. This may be the case in which the monitoring rule defines a relationship that must hold between data existing within the process in execution and data that can be obtained by interacting with external data collectors.

WS-CoL does not make use of keywords `\old` and `\result`[3]. The first is not useful because services are black-boxes that take input messages and produce output messages. Therefore, it is never necessary to refer to the value a certain "variable" possessed prior to the invocation of the operation. The second keyword is useless because we can refer to returned messages with their names.

[2] This is what the system produces but the user defines locations by pointing to the specific WS-BPEL elements directly in the graphical editor, and by choosing the annotation type.

[3] Lack of space does not allow us to thoroughly introduce the language, but JML uses `\old` to refer to old values in post-conditions, and `\result` to identify the value returned by a method.

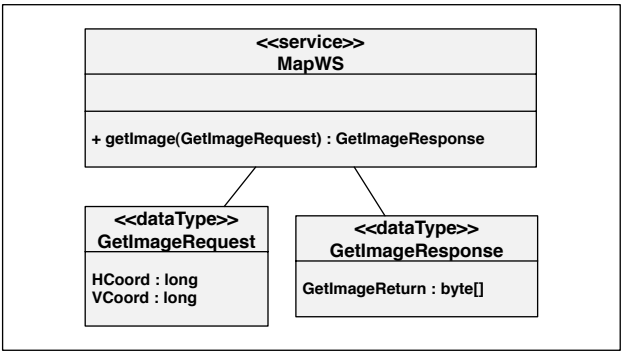

Fig. 5. The MapWS Web Service

WS-CoL adds a set of keywords that represent ways of obtaining data from external data collectors. A different extension is introduced for each of the standard XSD types that can be returned by external data collectors: `\returnInt`, `\returnBoolean`, `\returnString`, etc. Therefore, while defining a monitoring expression, we can use these extensions. All follow the same design pattern. They take as input all the information necessary for interacting with the external data collector, such as the URL location of its WSDL description, the name of the operation to be called upon it, the parameters to be passed to the data collector service, etc (see Section 4).

For example, if we want to specify a post-condition for the `getImage` operation in Figure 5 and state that the returned map must have a resolution less than "80x60" pixels we would define the expression as:

```
@ensures \returnInt(wsdlLoc, getResolution,
'image', GetImageResponse.GetImageReturn,
HResolution) <= 80 &&
\returnInt(wsdlLoc, getResolution, 'image',
GetImageResponse.GetImageReturn, VResolution) <= 60;
```

In this example, a `getResolution` operation is invoked on a service that publishes its interface at the URL `wsdlLoc`. The array of bytes `GetImageReturn` (see Figure 5) is passed as an input value and mapped onto the `image` message part defined at `wsdlLoc`. `HResolution` and `VResolution`, on the other hand, are the message parts defined in the output message at `wsdlLoc` that should be returned as integers. These returned values are compared with the desired resolution (80 pixels for the horizontal dimension and 60 pixels for the vertical dimension).

4 Monitoring Manager

The *Monitoring Manager* is the key component of our proxy-based solution for dynamic monitoring. This section illustrates its architecture and how it can be used by a WS-BPEL process that requires monitoring. We also analyze how its structure impacts the transformation produced by the BPEL^2 pre-processor.

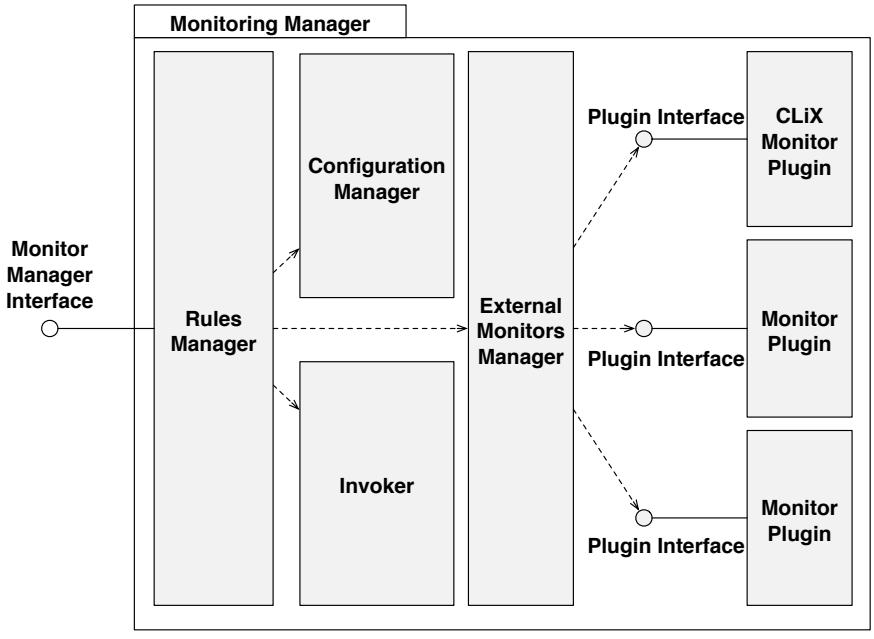

Fig. 6. The Monitoring Manager

The manager, whose architecture is shown in Figure 6, is capable of interpreting monitoring rules, of keeping trace of the configuration with which a user wants to run a process, of interacting with external data collectors to obtain additional data for monitoring purposes, and of invoking external monitor services.

We illustrate its use in the case of monitoring of pre- and post-conditions; its usage for the other cases is similar. To evaluate pre-conditions, the manager is used in substitution to the services which have rules associated with them. In fact, it is called *instead* of the service to be monitored. When called, it decides if the rule is to be evaluated by looking at its associated monitoring parameters and if it is, it proceeds to evaluate it. If the condition is verified correctly, it then invokes the original web service being monitored. Post-conditions are evaluated in the same way.

The manager is constructed to keep a configuration table for each process execution. These configurations are managed by the *Configuration Manager*. In particular, the manager needs to know: the initial overall process configuration (contained in the monitoring definition file), the monitoring rules, and all the information necessary for interacting with external services (the service being monitored, the external data collectors, and the external monitor service). Most of these data can be sent to the manager at the beginning of the process by invoking the setup method published by the manager (see Figure 1). In particular, everything except the input/output values that will be exchanged at run-time can be sent at the beginning of the process, before starting to perform the real business logic. This solution is preferable, with respect to sending all the data every time the process needs to interact with the manager, since an initial slowdown is certainly better than slowing down all the intermediate steps. All the in-

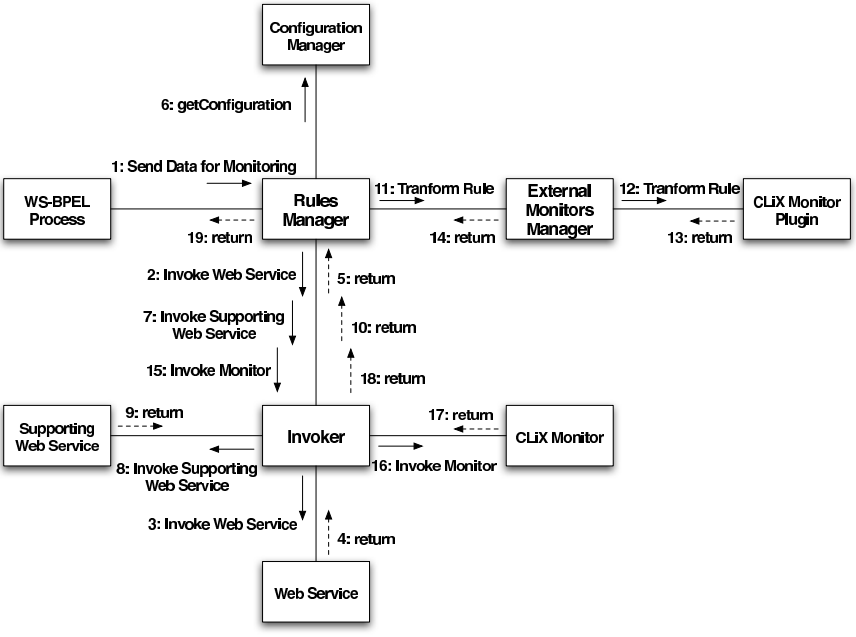

Fig. 7. The Monitoring Manager

formation sent during the setup phase is stored in the *Configuration Manager* and is associated with a process execution through the unique identifier produced by the WS-BPEL engine. Similarly, at the end of a process execution the manager is warned to free itself of the burden of keeping the corresponding configuration table.

The manager also supplies a graphical interface to the user. It permits the run-time consultation and modification of the values contained in the configuration table. For example, it is possible to modify the priority level at which a process is being run or to add a new provider to the list of certified providers that are associated with a given monitoring rule.

Figure 7 shows the step by step interaction of the components that cooperate to execute the service presented in Section 3 and to check its post-condition[4]. Initially, the BPEL process sends the data that will be necessary to the manager (Step 1). Since no pre-condition needs to be checked, the *Rules Manager* asks the *Invoker* to go on and invoke the external web service (in our case service `MapWS`) (Steps 2 and 3). When the *RulesManager* receives the results of the service invocation (Steps 4 and 5), it interacts with the *Configuration Manager* to retrieve the monitoring rule (i.e. the post-condition) that has to be checked (Step 6). By examining the monitoring parameters attached to the rule, the *Rules Manager* dynamically decides if the rule is to be checked or not. For example, if we consider the expression presented in Section 3, we could imagine the

[4] More complete running examples are available at : `http://www.elet.polimi.it/upload/guinea`.

associated priority parameter to be 4. If the process is then run with a priority value of 3, the rule would be checked since its priority parameter is higher than the value associated with the process.

Then, *Rules Manager* decides whether additional data are required from external data collectors. If this is the case, it calls the *Invoker* to obtain them (Step 7). This component is built around Apache WSIF (Web Service Invocation Framework [3]) and is capable of invoking a web service without previously creating client-side stubs but by dynamically interacting with the service through its WSDL description. The *Invoker* can be used to invoke any service provided it knows: the URL of the WSDL of the service to be invoked, the name of the operation that is to be invoked on that service, a list of keys that help map the operation's input values onto the operation's message parts as defined in the WSDL description, a list of input values for the operation to be invoked, and a list of keys for indicating the parameters (as indicated in the output message parts contained in the WSDL description) we want to receive as output. The *Invoker* can also be called when an expression uses a WS-CoL to obtain additional monitoring data from external data collectors. In this case, the list of output keys is reduced to a single key that corresponds to a part of the output message as described in the WSDL description of the service (see the expression given in Section 3.1).

Once all the data necessary have been obtained (Steps 8, 9, and 10), the *RulesManager* begins its interaction with the *External Monitors Manager* (Step 11). This component is responsible for managing the different kinds of external monitors that the *manager* is capable of working with. In particular, it manages the set of plugins that contain the logic necessary for converting the WS-CoL syntax used for defining the monitoring expressions into the proprietary syntax used by each external monitor. The monitor plugin also prepares the data that must be sent to the monitor by formatting them in a way that the monitor is capable of interpreting (Step 12). In this paper, we use a monitor built around `XlinkIt` [1]. For this monitor the WS-CoL expressions must be re-written as CLiX rules and the data expressed as XML fragments. When the *External Monitors Manager* has finished adapting the monitoring data and the monitoring rules (Steps 13 and 14), the *Invoker* is called once again for invoking the external monitor (Step 15). If the monitor responds with an error, meaning the condition is not satisfied, the *Rules Manager* communicates it to the WS-BPEL process by returning a standard fault message, as published in the WSDL description of the manager. If the monitor's response is that the condition is satisfied, the manager can then proceed to return the original service response to the *WS-BPEL Process* (Step 19).

5 Related Work

The research initiatives undertaken in the field of web service monitoring share the common goal of discovering erroneous situations during the execution of services. They differ, although, in a number of ways: degree of invasiveness, abstraction level at which they work, reactiveness or pro-activeness.

For example, Spanoudakis and Mahbub [9] developed a framework for monitoring requirements of WS-BPEL-based service compositions. Their approach uses event-calculus for specifying the requirements that must be monitored. Requirements can be

behavioral properties of the coordination process or assumptions about the atomic or joint behavior of deployed services. The first can be extracted automatically from the WS-BPEL specification of the process, while the latter must be specified by the user. Events are then observed at run-time. They are stored in a database and the run-time checking is done by an algorithm based on integrity constraint checking in temporal deductive databases. Like our approach, it supports reactive monitoring since erroneous situations can be found only after they occur, but it is less intrusive since it proceeds in parallel with the execution of the business process. This leads to a lesser impact on performance but also to a lesser responsiveness in discovering run-time erroneous situations. The approach also proposes a lower abstraction level, placing therefore a heavier burden on the designer.

Lazovik et al. [10] proposes another approach based on operational assertions and actor assertions. The first can be used to express properties that must be true in one state before passing to the next, to express an invariant property that must hold throughout all the execution states, and to express properties on the evolution of process variables. The second can be used to express a client request regarding the entire business process, all the providers playing a certain role in the process execution, or a specific provider. The system then plans a process, executes it, and monitors these assertions. This approach shares with ours the fact of being assertion-based. Once the assertions are inserted, it is completely automatic in its setup and monitoring. It lacks although the possibility of dynamically modifying the degree of monitoring. It also lacks adoptability since it is based on proprietary solutions.

Our approach must also be compared with the proposals that integrate Aspect Oriented programming and WS-BPEL. An example can be found in the work by Finkelstein et al. [5]. It exploits the semantic analyzers present in their development toolkit (called SmartTools) to implement a WS-BPEL engine as an interpreter. Abstract syntax trees are built for each process and are then traversed by the semantic analyzer that implements the visitor design pattern. These methods facilitate aspect oriented adaptation. The approach concentrates more on weaving at the engine level and less at the process level, which is where our approach works.

6 Conclusions and Future Work

The paper has presented an approach to support the *dynamic monitoring* of WS-BPEL processes. It is an evolution and refinement of the ideas already presented in [7]. The proxy-based solution is dictated by the wish of using available technology, instead of inventing new non standard executors, but this proposal can also be seen as a feasibility study to better understand the different pieces of the approach, and evaluate the possibility of embedding them in an existing WS-BPEL engine.

Our future work will concentrate on further studying the possibility of embedding the *monitoring manager* into a WS-BPEL engine, on experimenting with new *data collectors* and *data analyzers*, on extending the language to support other types of monitoring (e.g., the capability of predicating on histories instead of concentrating on punctual values), and on providing real-world results of the performance ”overhead” that can be introduced by our approach.

References

1. XlinkIt: A Consistency Checking and Smart Link Generation Service. *ACM Transactions on Software Engineering and Methodology*, pages 151–185, May 2002.
2. AXIS. Apache AXIS Web Services Project, 2005. `http://ws.apache.org/axis/`.
3. Web Service Invocation Framework. Apache WSIF Project, 2005. `http://ws.apache.org/wsif/`.
4. D.C. Luckham. Programming with Specifications: An Introduction to Anna, A Language for Specifying Ada Programs. *Texts and Monographs in Computer Science*, Oct 1990.
5. C. Courbis and A. Finkelstein. Towards Aspect Weaving Application. *In Proceedings of the 25th International Conference on Software Engineering*, 2005.
6. N. Delgado, A.Q. Gates and S. Roach. A Taxonomy and Catalog of Runtime Software-Fault Monitoring Tools . *IEEE Transactions on software Engineering*, pages 859-872, December, 2004.
7. L. Baresi, C. Ghezzi and S. Guinea. Smart Monitors for Composed Services. *In Proceedings of the 2nd International Conference on Service Oriented Computing*, 2004.
8. L. Baresi, C. Ghezzi and S. Guinea. Towards Self-healing Service Compositions. *In Proceedings of the First Conference on the PRInciples of Software Engineering*, 2004.
9. K. Mahbub and G. Spanoudakis. A Framework for Requirements Monitoring of Service Based Systems. *In Proceedings of the 2nd International Conference on Service Oriented Computing*, 2004.
10. A. Lazovik, M. Aiello and M. Papazoglou. Associating Assertions with Business Processes and Monitoring their Execution. *In Proceedings of the 2nd International Conference on Service Oriented Computing*, 2004.
11. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. *Department of Computer Science, Iowa State University, TR 98-06-rev27*, April, 2005.