Integrated and Composable Supervision of BPEL Processes

Luciano Baresi, Sam Guinea, and Liliana Pasquale

Politecnico di Milano - Dipartimento di Elettronica e Informazione
via Golgi, 40 – 20133 Milano, Italy
{baresi,guinea,pasquale}@elet.polimi.it

Abstract. In the past few years many supervision techniques were developed to provide reliable business processes and guarantee the established SLAs. Since none of them provided a definitive solution, the paper proposes a new composable approach, where a single framework provides the glue for different probing, analysis, and recovery capabilities. The paper introduces the framework and exemplifies its main features.

1 Introduction

Since the uptake of service technology as a means to develop complex distributed systems, *monitoring* and *recovery* tools have played a very significant role. In this paper we refer to the combination of the two as *supervision*, and concentrate on systems designed using BPEL (Busines Process Execution Language [4]).

In the past years many supervision techniques were developed, with different requirements they deem important. As for monitoring, the authors themselves have proposed two very different solutions. The first is Dynamo [3], a synchronous and assertion-based approach to monitoring. Even if it is very intrusive, since the process is blocked every time it performs a service invocation, it is able to discover anomalous behaviors as soon as they occur. The second is ALBERT [1], an asynchronous approach based on temporal logic. It is less intrusive, since all the assertions are checked in a separate thread but it can capture anomalies only when the process has already proceeded beyond the point in which they were generated. Other works, like VieDAME [7], check non functional properties (e.g., response time, accuracy) by analyzing the messages the process exchanges with partner services. If we move to recovery, the authors have tackled it with WSReL [2], which offers a set of pre-defined atomic recovery actions (e.g., service substitution, email notification, rollback), that can be mixed to create complex recovery strategies. Other approaches focused on service substitution, applying dynamic binding techniques, as proposed by Colombo et al. [5], or using an AOP-based extension of ActiveBPEL, as proposed by Moser et al. [7].

Although each of these approaches is particularly effective in its own subdomain, none of them provides a holistic solution that easily accommodates all different clients, in terms of qualities of interest and required analysis. Instead of searching for one definitive solution, we provide an integrated framework in

A. Bouguettaya, I. Krueger, and T. Margaria (Eds.): ICSOC 2008, LNCS 5364, pp. 614–619, 2008.

which diverse solutions can be combined to exploit their main advantages and meet different users' needs.

Our vision of a unified framework builds upon the decoupling of data collection, monitoring, and recovery. Data collection is independent of the types of supervision approaches combined. Monitoring uses these data to check functional and non-functional properties, while recovery uses them, together with the monitoring results, to attempt to fix the process, produce a log, or perform post-mortem activities to prevent them from happening again.

The paper is organized as follows. Section 2 presents our integrated framework and shows. Section 3 describes some example rules for the interplay of data collection, monitoring, and recovery. Section 4 concludes the paper.

2 Integrated Supervision Framework

When conceiving our framework, we wanted to satisfy different requirements. It had to support different quality dimensions and different analyses. Synchronous analysis can be used to evaluate punctual process properties (e.g., the response time of partner services). Asynchronous analysis can be exploited to measure temporal process properties (e.g., the number of times a synchronous check is violated). Post-mortem analysis can be used to construct a symptom model of process failures. Our framework should apply suitable recoveries with different timeliness depending on the analysis that signaled the violation.

The distinction among data collection, monitoring and recovery allowed us to conceive a neater architecture. Data collection fosters the neat separation between monitoring and recovery activities. We support *internal data*, which carry the internal state of the process, *external data*, which provide information from the surrounding environment, and *historical data*, which represent information collected in past executions. Different monitoring approaches are also able to share partial results and collaborate towards a more complete final assessment. Our framework can trigger corrective actions on the same process instance, on different instances (of different processes), and also on the process definition. To this end, we allow the interplay between synchronous and asynchronous actions and we provide conflict resolution mechanisms associating them to a priority.

Figure 1 shows the overall architecture of our solution. Each *BPEL Engine* is an instance of *ActiveBPEL Community Edition Engine* augmented with AOP (aspect-oriented) probes to collect process state data. The *Data Manager* is responsible for collecting external data, and for retrieving and storing historical data from/in the *Data Repository*. The *Monitoring Farm* holds the monitoring plug-ins we want to use, while the *Recovery Farm* holds the recovery capabilities.

The *Event Controller* is the central element of our architecture and it is based on rule engine technology [6]. It is in charge of activating external and historical data collection, as well as any monitoring and recovery activity. While internal variables are passively received from the AOP probes embedded in ActiveBPEL, external and historical variable collection, like also monitoring and recovery activities, are triggered when the process starts or when it reaches a particular

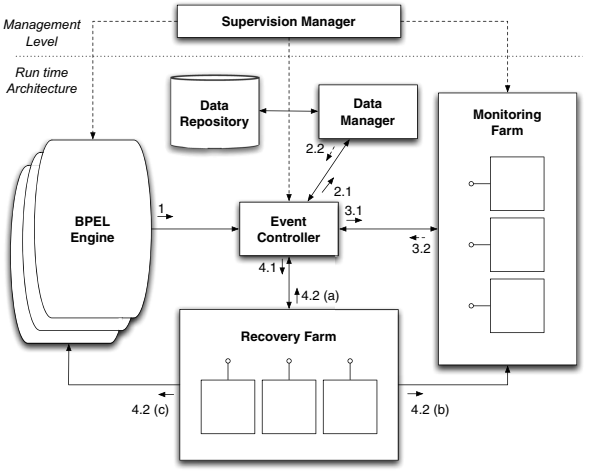

Fig. 1. The Unified Framework

state. This is achieved by defining rules on the data contained in the working memory, that is, process state data, external and historical variables, collected through the *Data Manager*, and analysis results.

Our framework also provides a configuration tool called *Supervision Manager*, used to configure the various components of the framework. In particular, it configures the AOP probes to collect internal data, the *Event Controller*, to retrieve external and historical data, and defines how monitoring and recovery are activated. Finally, it also sets the *Monitoring Farm* with the constraints each plug-in is supposed to check.

The numbered arrows of Figure 1 explain how the framework works. Before starting the execution of any process, the *Supervision Manager* configures the different components to allow them to correctly perform supervision tasks. After this, every time an executing process terminates an activity, the AOP probes collect internal data and give them to the *Event Controller* (transition 1), which inserts them in its embedded working memory. The data available in the working memory of the *Event Controller* can always activate suitable rules to require the *Data Manager* to retrieve external or historical data (transition 2.1) and store them in its working memory (transition 2.2).

3 Example Configuration Rules

This Section shows some configuration rules to exemplify how the framework works. The following rule shows how to collect an external variable.

```
rule "external_data_collection"
no-loop true
when
```

```
 p : ProcessState(process = "SMS_Process",
      engineID = "localhost:8080", instanceID = -1,
      username = "jack.burton", /process/sequence/Invoke_SMS_Send)
then
 ExternalVariable e = DataCollector.pull(wsdl, operationName, input);
 insert(e);
```

The external variable is collected when process `SMS_Process` reaches activity `/process/sequence/Invoke_SMS_Send` and `ProcessState p` appears in the working memory. Since we currently support external probes implemented as Web services, we need the endpoint reference (*wsdl*), the name of the operation to invoke (`operationName`), and its input parameters (`input`). The datum is then inserted in the working memory through operation `insert`, defined in the *then* clause. Notice that the input parameters given to an external probe can include any data present at that time in the working memory of the *Event Controller*.

Besides data collection and storage, rules are also responsible for dispatching data to interested monitoring plugins (transition 3.1). This feeds the creation of special dispatching rules in the *Event Controller*'s rule engine. For each monitoring plugin being used, designers must then express the actual assertions they want them to check. Each expression is given in the language used by the plugin.

The following rule uses ALBERT to asynchronously check whether the accuracy of a service endpoint is greater than a predefined threshold. It passes ALBERT all necessary data to perform monitoring: the current partner service endpoint (variable `pLink`) and its current accuracy (variable `mResult`) calculated by VieDAME. Variable `pState` detects the process location in which the monitoring result and the service endpoint need to be forwarded.

```
rule "ALBERT data"
when
 pState : ProcessState(processName == SMS_Process,
      location == "/process/sequence/Receive_SMS_Notification") &&
 pLink : PartnerLink(processName == SMS_Process,
      location == "/process/sequence/Receive_SMS_Notification",
      pLinkN : name == "SMS_sending")
 mResult : MonitoringResult(
      pluginName == "VieDAME", n : rule == "accuracy",
      processName=="SMS_Process",
      location == "/process/sequence/Receive_SMS_Notification")
then
 List<Datum> data = new ArrayList<Datum>();
  data.add(n, mResult);
  data.add(pLinkN:pLink);
 dispatchData("ALBERT", pState, data);
```

ALBERT checks the following rule:

```
processName: SMS_Process
rule: onEvent(/process/sequence/Receive_SMS_Notification) ->
    $accuracy/partnerLink[@name == $SMS_sending/name]
      /endpoint[@value == $SMS_sending/address]/accuracy > THRESHOLD;
```

where `$accuracy` contains the monitoring results provided by VieDAME, while `$SMS_sending` contains the partner link being used by process `SMS_Process`.

As soon as a monitoring result is produced by the *Monitoring Farm*, it is sent to the *Event Controller* to insert it in its working memory (transition 3.2). These new data can fire rules that request the *Recovery Farm* to apply some recovery actions (transition 4.1) to modify the executing process instances, change how the *Event Controller* works, or modify the checks performed by the monitoring plug-ins (transition 4.2). At this point, the *Event Controller* has all the monitoring results and can activate different recovery strategies by communicating with the *Recovery Farm*. The recovery plugins in the *Recovery Farm* can access the process internals using AOP probes. In our current implementation we provide the same set of recovery actions defined in [2], to tune the overall degree of monitoring, both in terms of activities being performed and approaches being used, and also asynchronous recovery, to work on process definitions directly and change how future process instances are configured.

Synchronous recovery needs the process to be blocked the process execution until it completes, while asynchronous recovery can be performed in parallel with the process activities till the process reaches a specific target location. In has to stop until recovery completes. If there are multiple asynchronous recoveries ready to be executed, the *Event Controller* chooses among them depending on their priority level as well as on the strategy the framework must use to interpret these priority values. For example, if the designer decides to use an *exclusive* strategy, a recovery with the highest priority disables all the others. This strategy is realized by assigning the same activation-group attribute to potentially conflicting rules. The activation-group attribute guarantees that only the rule with highest priority is executed. If the designer decides to go with an *all* strategy, all recoveries are executed and the order is given by their priority values. Recovery strategies with the same priority level can execute in any order. If the designer decides not to provide priorities, recovery strategies can be executed in any order.

The following rule defines a recovery for when the previous ALBERT rule is not verified (variable `malbert`). This recovery has the effect of changing the process definition, substituting a partnerLink with the one suggested by VieDAME to have the best accuracy (variable `mviedame`).

```
rule 'SMS_asynch_substitute'
activation-group "async_substitution"
salience 2
  when
    malbert : MonitoringResult(plug-in == "ALBERT",
         processName == "SMS_Process",
         location == "/process/sequence/Receive_SMS_Notification",
         result == false) &&
    mviedame : MonitoringResult(plug-in == "VieDAME",
         type == "bestAccuracy"
         processName=="SMS_Process",
         location == "/process/sequence/Receive_SMS_Notification")
  then
```

```
recovery.setProcess('SMS_Process');
recovery.setTargetState('/process/sequence/Invoke_SMS_Service');
recovery.asynchSubstitute(mviedame.getPartnerLinkName,
 mviedame.getWsdl(), mviedame.getOperation(), mviedame.getInput(),
  mviedame.getTransformationRule());
```

The asynchronous recovery is assigned a priority of 2 (`salience`) and an activation-group named `async_Substitution`, and it is configured to execute before activity `/process/sequence/Invoke_SMS_Service`. The recovery consists in an invocation of method `asynchSubstitute` provided by our framework. We pass to the recovery method the information gathered from ViEDAME, as well as an indication of the process on which the recovery has to take place (`SMS_Process`).

4 Conclusions and Future Work

We propose a flexible and customizable supervision framework for BPEL processes, integrating different monitoring and recovery techniques. The framework architecture conceptually decouples data collection, monitoring, and recovery. Leveraging rule engine technology we dispatch data to various monitoring components, aggregate different monitoring results, and choose the most appropriate recovery. In our future work, we will investigate high level supervision languages and tools that can help hide the complexities that lie in the definition of the rules used by our *Event Controller*. We also want to discover suitable ways to synthesize these rules starting from requirements and SLA standards.

References

1. Baresi, L., Bianculli, D., Ghezzi, C., Guinea, S., Spoletini, P.: Validation of Web Wervice Compositions. IET Software 1(6), 219–232 (2007)
2. Baresi, L., Guinea, S.: A Dynamic and Reactive Approach to the Supervision of BPEL Processes. In: Proceedings of the 4th India Software Engineering Conference (2008)
3. Baresi, L., Guinea, S.: Towards Dynamic Monitoring of WS-BPEL Processes. In: Proceedings of the 3rd International Conference on Service Oriented Computing, Amsterdam, The Netherlands, December 12-15, pp. 269–282
4. OASIS, Business Process Execution Language for Web Services, Version 1.1
5. Colombo, M., Di Nitto, E., Mauri, M.: SCENE: A Service Composition Execution Environment Supporting Dynamic Changes Disciplined Through Rules. In: Proceedings of the 3rd International Conference on Service Oriented Computing, Chicago, IL, USA, December 4-7, 2006, pp. 191–202 (2006)
6. Proctor, M., et al.: Drools, `http://www.jboss.org/drools/`
7. Moser, O., Rosenberg, F., Dustdar, S.: Non-intrusive monitoring and service adaptation for WS-BPEL. In: Proceedings of the 17th International Conference on World Wide Web, Beijing, China, April 21-25, 2008, pp. 815–824 (2008)