Self-Healing BPEL Processes with Dynamo and the JBoss Rule Engine

Luciano Baresi, Sam Guinea, and Liliana Pasquale
Dipartimento di Elettronica e Informazione Politecnico di Milano
piazza Leonardo da Vinci, 32
Milano, Italy
guinea@elet.polimi.it

ABSTRACT

Many emerging domains such as ambient intelligence, context-aware applications, and pervasive computing are embracing the assumption that their software applications will be deployed in an open-world. By adopting the Service Oriented Architecture paradigm, and in particular its Web service based implementation, they are capable of leveraging components that are remote and not under their jurisdication, i.e. services. However, the distributed nature of these systems, the presence of many stakeholders, and the fact that no one has a complete knowledge of the system preclude classic static verification techniques. The capability to "self-heal" has become paramount.

In this paper we present our solution to self-healing BPEL compositions called Dynamo. It is an assertion-based solution, that provides special purpose languages (WSCoL and WSReL) for defining monitoring and recovery activities. These are executed using Dynamo, which consists of an AOP-extended version of the ActiveBPEL orchestration engine, and which leverages the JBoss Rule Engine to ensure self-healing capabilities. The approach is exemplified on a complex case study.

1. INTRODUCTION

Recently, many evolving domains, such as ambient intelligence, context-aware applications, and pervasive computing, have turned towards service-oriented architectures as an enabling paradigm, and amongst the possible technological implementations, Web services have emerged as the leading solution. A distinct characteristic —shared by all these domains— is that they all perform an important assumption-shift regarding the surrounding world. The applications we want to build are intrinsically distributed, and extremely dynamic, and we want to harness the possibilities the environment is capable of offering us.

When adopting a web-based service oriented paradigm, we build applications (usually using composition techniques such as BPEL [14]) that leverage remote services that we do not own. The distributed nature of our applications, the intrinsic high level of dynamism and flexibility, and the fact that we do not own all of the system's components, preclude the complete pre-deployment validation of the application. This leaves us in an uncertain situation; a number of unexpected and potentially catastrophic events may arise. Examples of such events are *services not responding or responding too slow*, *external technological failures*, or *functional and quality of service (QoS) misbehavior*. In the first case we have a service that fails to respond. There are many reasons why a service might not respond. For example, the problem might be due to a network connection failure, or to the service being un-deployed by its provider. Timeouts provide a good approximation for catching services that do not answer. In the second case, we assist to implementation or protocol errors, while in the third we are confronted with services that fail to provide the functional and/or non-functional qualities we need.

This is why we see an increasing interest towards *self-healing* solutions, both in industry and in research, and in particular towards systems that *can detect faults and errors instantly and then contain the effects of the faults within defined boundaries. This allows applications to recover from the negative effects of such anomalies executing compensation actions* [13].

Some of the authors have already obtained important results in the field of self-healing Web services [4]; these results have converged into a supervision framework for BPEL processes called Dynamo [3, 5]. In this paper, we propose a holistic approach for the addition of self-healing capabilities to BPEL processes. We assume that the process is internally consistent, and that erroneous situations can only arise during the interaction with partner services. This is why we propose to complete process designs with a declarative indication of the functionality and QoS the external services are supposed to guarantee. We use pre- and post-conditions —specified using our special-purpose languages WSCoL and WSReL— on the process' *invoke*, *receive*, and *pick* activities. We also support invariants over BPEL scopes through appropriate translation into post-conditions. During execution, we check these assertions process-side and react to anomalies by supplementing the process with self-healing capabilities.

The goal of this paper is twofold. First of all, we illustrate our approach in the context of a complex case study. This gives us the possibility to illustrate the advantages of our solution, and how it can be effectively used in real scenarios. Secondly, it introduces a completely new implementation of the Dynamo framework, built using the JBoss Rule Engine [16], and clarifies the advantages of such an implementation. The choice to implement Dynamo on top of the JBoss Rule Engine ensures the following characteristics:

- *Logic and data separation.* Data are incapsulated in domain objects, while the logic is in the rules. The upshot is that organizing logic in a multi-layer fashion eases its maintenance with respect to changes.
- *Speed and scalability.* The algorithm for verifying the rules (Rete) provides very efficient ways for matching rule patterns and our domain object data. The rule engine can manage a big number of rules with minimum impact on the normal execution flow of the process.
- *Knowledge centralization.* The rule engine creates a knowledge base, which is executable, creating a central point of truth for the whole business policy.
- *Tool integration.* Tools like *eclipse* provide ways to edit and manage rules and get immediate feedback and validation. Auditing and debugging tools are also available.[16]

The rest of the paper is organized as follows. Section 2 introduces the case study and informally identifies the functionalities and QoS that need to be guaranteed. Section 3 presents the overall approach for adding self-healing features. Section 4 uses WSCoL and WSReL to formally define the supervision activities performed by Dynamo. Section 5 presents the new implementation of the supervision framework based on JBoss Rules. Section 7 surveys some related approaches and Section 8 concludes the paper.

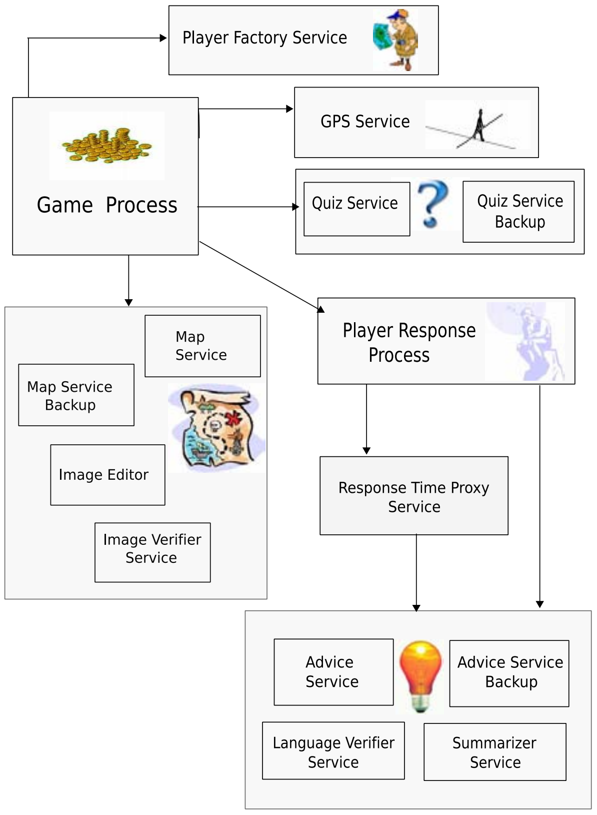

Figure 1: Game Process.

2. CASE STUDY

To understand the features and capabilities of the self-healing framework we exemplify them on a treasure hunt. The goal of this game is to be the first to find a treasure. Each player is given a GPS-enabled PDA. By roaming through the environment, the players can reach intermediate checkpoints in which they are asked to answer a simple question. If they answer correctly, the location of the next checkpoint is revealed on a map in their PDA. If players answer incorrectly, they loose 10 points, and have to answer a new question. They players do not receive a new map until they have answered correctly at least once. If a player looses all his points, he must leave the game. Players can also ask for suggestions, using an appropriate service. For each suggestion the player must pay 1 point. At the beginning of the game each player has 30 points and is given the location of the first checkpoint.

Figure 1 shows the centralized BPEL process, called *Game Process*, that manages the game. It is in charge of interacting with many different services and in particular with: *Player Factory Service*, to register the players, *GPS Service*, to keep track of each player's current position in the form of GPS coordinates and warn the BPEL process each time a player reaches a checkpoint, *Quiz Service*, to query each player at the various checkpoints, *Map Service*, to provide the maps for the next checkpoint, and *Player Response Process*, to control the interactions between the BPEL process and a player. This last service is also modeled as a BPEL process, and interacts with *Advice Service* to request suggestions.

The example also requires a number of auxiliary services that are not used directly by the *Game Process*, but that will be used to recover from anomalous situations: a *Language Verifier Service* is capable of detecting the language used in a given sentence, an *Image Verifier Service* can calculate a map's resolution, an *Image Editor Service* is able to resize an image, a *Summarizer Service* summarizes sentences, and a *Response Time Proxy Service* measures the average response time of the different *Advice Services* (in our scenario we have more than one of these, allowing us to switch dynamically between them if needed). This service also keeps track of the fastest.

Plausible anomalies in this scenario could be: (a) the *Advice Service* does not provide a suggestion within 5 seconds, (b) the *GPS Service* provides coordinates in a wrong format, (c) the *Quiz Service* returns a question in a language different from that of the player, or (d) the *Advice Service* produces an advice that is either too long or too short for the *Player Response Process*. Another constraint is that each player must always have more than zero points.

In our case, the key components, (*Game Process* and *Player Response Process*) must guarantee their correct functionality and maintain the promised levels of QoS. Classical pre-deployment techniques loose effectiveness in such a system and a good mix of probing and reactions is the only way to keep the execution on track.

3. SELF-HEALING CAPABILITIES

Our proposal for *self-healing* BPEL processes is based on the use of two special-purpose languages: *WSCoL*, for specifying constraints on the execution of BPEL processes, and *WSReL*, to state recovery strategies that should be executed when constraints are violated. The orthogonality between these languages and BPEL allows the designer to keep the definition of the business problem and its self-healing features separate, therefore limiting unforeseen side effects.

These languages are highly flexible and customizable. It is possible for users to customize monitoring and recovery actions depending on their needs, and depending on the reliability level required. For example, the prioritization and the definition of different recovery strategies for the same violated constraint can trigger specific recovery actions in different contexts. The level of monitoring can also be modified at runtime and users are always free to dynamically add new constraints or recovery strategies to their processes. Monitoring and recovery activities are transparent to users: they only see their original 100% pure BPEL processes, suitably augmented with self-healing directives.

More specifically, our *self-healing* capabilities are materialized by adding *supervision rules* to the BPEL process. Each rule comprises:

- A **location** that defines the exact point of the process in which we want a certain property to be verified, in other words it contains an XPATH that points to the location of the BPEL activity in which the monitoring rule has been defined and a keyword that specifies if the rule is a pre-condition, a post-condition, or an invariant. It also defines where we want to apply recovery strategies if needed.

- Some **monitoring parameters** that tailor the degree of supervision we want to achieve at run-time by allowing the designer to define the cases in which the rule can be ignored during execution. Supported parameters are priority, validity, and trusted providers. Priority represents the importance of a rule with respect to the BPEL process[1]. The validity defines a time window in which the monitoring rule is checked. Trusted providers is a list of service providers for which the designer decides that invocations can go unmonitored.

- A **monitoring expression** that defines the constraints that must hold at run time, that is, the core of the monitoring logic. This expression is stated in WSCoL.

- A **recovery strategy** that defines what the system should do to try to keep things on track. These strategies are defined in WSReL.

Lack of room does not allow to fully describe the two special-purpose languages, thus we prefer to introduce their main features and refer the readers to [10] for an in depth presentation of the two languages.

[1]Notice that the process is executed with a process-wide priority that defines a threshold indicating which rules can be ignored.

3.1 WSCoL

WSCoL (Web Service Constraint Language) provides language specific construct for data collection and data analysis.

Data collection is responsible for obtaining the monitoring data that are used to check whether remote services behave consistently with defined assertions (conditions). WSCoL distinguishes between three kinds of monitoring data: internal, external and historical variables. *Internal variables* consist of data that belongs to the persistent state of an executing process. For each activity it is only possible to refer to the variables used by the activity itself, or that are defined within some recursively enclosing scopes. *External variables* consist of data that cannot be obtained from the process in execution, but can be obtained externally from any remote component provided it exposes a WSDL interface. This means that external variables allow us to embed probe Web services directly in the WSCoL expression. External services are useful, for example, when the correctness of a condition depends on certain QoS properties which can only be collected through special suitable probes. *Historical variables* consist of monitoring data collected during previous process executions. WSCoL allows designers to store variables into a persistent storage component contextually to the analysis of a set condition. A historical variable is therefore implicitly tied to the process definition and to the process instance it belonged to, as well as to the condition in which the storage operation was performed. During data collection it is also possible to create aliases. This feature provides two fundamental advantages: they result in a value being collected only once and they allow us to create more compact analysis and recovery statements.

Data Analysis defines the functional and non functional correctness of an external invocation by stating the relationships that must hold among collected data. WSCoL supplies typical boolean operators such as && (and), || (or), ! (not), $\Rightarrow$ (implies), and $\Leftrightarrow$ (if and only if), typical relational operators, such as $<$, $>$, $==$, $\leq$, and $\geq$, and typical mathematical operators such as $+$,$-$,$*$, $/$, and %. The language also allows us to predicate on sets of values through the use of universal (*forall*) and existential (*exists*) quantifiers, and of other constructs such as *max*, *min*, *avg*, *sum* and *product*. These constructs become quite meaningful in conjunction with historical variables. They allow designers to compare the behavior of a remote service with previous iterations, to discover, for example, if there has been significant performance deterioration with respect to the past.

3.2 WSReL

WSReL (Web Service Recovery Language) is used for specifying recovery strategies. Our approach is intrinsically synchronous and the process is momentarily blocked while recovery is executed. Recovery has no power over the process definition itself, but recovery strategies have impact only on single process instances that belong to the process definition for which they supervision rules are created. Moreover users are able to enable or disable recovery activities through the prioritization mechanism already introduced.

WSReL provides a programmable, flexible, and extensible solution to the recovery problem: users can define their own recovery strategies by mixing atomic actions. These actions can be divideqd in two groups: recovery actions that terminate successfully by definition, and those that need to be re-verified to assert success.

The first group comprises: *ignore*, to simply ignore the fact that an anomaly has arisen, *notify*, to send an email message to inform the stakeholder that something wrong has happened, *halt*, to stop the process in execution, *call*, to invoke an external Web service, which does not share the same data space as the process, *substitute*, to replace a service invocation with an invocation to a different service that provides the same functional and non-functional properties, and *callback*, to directly call an event handlers embedded in the BPEL process.

Action *call* is useful since it gives users great flexibility. For example, a designer can statically devise an external BPEL process that uses runtime information (coming both from the process and from the monitoring activities) to provide an augmented rebinding where UDDI registries are queried on-the-fly for appropriate candidates. In contrast, action *callback* presents some disadvantages: event handlers must be statically embedded into the process and the designer has to define them prior to deployment. This means that the recovery logic cannot be modified at runtime: the only dynamism allowed is through the parameterization of the event handler itself.

The second group of actions, i.e., those that require a further monitoring step, comprises: *retry*, to re-invoke the same Web service up to a certain number of times, *rebind*, to substitute the current service with another one that implements the same WSDL interface, and whose URI is passed as a parameter, *changeMonitoringRules*, to allow the designer to modify the amount of monitoring and therefore to relax or tighten the constraints, and *changeProcessParams*, to modify the monitoring parameters associated with the executing process. This allows for the dynamic modification of the supervision activities.

Recovery Strategies adopt the ECA rule paradigm (i.e., event - condition - action). The event consists in the violation the monitoring expression, the condition consists of nested *if-then-else* statements, and the action consists of a composition of atomic recovery actions. Strategies define exactly what the recovery subsystem should do to assure that the process execution remains on track. Complex strategies are combinations of strategy steps, which are in turn conjunctions of atomic recovery actions. Strategy steps are separated by the traditional *or* symbol (||), and the order in which they are specified is important. In fact, when we write `strategy_step1 || strategy_step2 || ...` we intend that the system first applies step 1 and, if that is not effective, tries step 2, and so on. If no step is successful, the process provider is notified and the process is halted.

On the other hand, a single strategy step is a conjunction (&&) of atomic recovery actions, meaning they all need to be performed. The success of a single strategy step depends on the success of the actions it contains. If a strategy step contains actions that do not need re-evaluation, this step is always considered successful. While if the strategy step contains actions that need re-evaluation, another monitoring phase is necessary to know whether the strategy step was successful or not. Generally, strategy steps with actions that are considered successfully by default are the last step of a recovery strategy.

4. EXAMPLE RULES

In the treasure hunt game previously explained, each player wants to reach the treasure as soon as possible. This means that the map must have a good PDA resolution, the services providing suggestions need to be fast and should always provide the right advice, each player needs to understand the language in which s/he is shown the question, and the advice must be neither too long nor too concise (in terms of number of words). Moreover we can also say that the coordinates given by the `GPS Service` must be in the correct format and each player must always have a number of points greater than 0. Given these requirements, we can detect three types of rules:

1. Rules related to the response time of a service: for example, the average response time of `Advice Service` must be less than (or equal to) 5 seconds.
2. Rules related to external errors: for example, `GPS Service` must work correctly and provide GPS coordinates in the right format.
3. Rules related to assertion infringements: for example, the language of a question must be the same as the language of the player, or each player must always have more than 0 points.

All these constraints, and many more, can be properly written in WSCoL. To constrain the average response time of `Advice Service` to be smaller than 5000 ms (i.e., 5 secs), we use historical data to compute the average response time of already performed invocations. For this reason, at the end of each invocation we store the response time of the service. If the service is not fast enough, `Player Response Process` should invoke `Response Time Proxy Service`, which continuously measures the response time of the various substitute advice services (including `Advice Service`, which is the default one) and thus knows the current fastest advice service. This implies that, during a recovery strategy, `Response Time Proxy Service` can re-invoke the same `Advice Service` used by `Player Response Process`. To formally render this, we start with the following supervision rule (from now on called *Average*) associated with the invocation of `Advice Service`:

```
precondition:
avg($rt in retrieve(processID, userID, instanceID, 0,
    /input/location, $response_time, 20); $rt) < 5000;

recovery strategy:
call(ResponseTimeProxyService_WSDL) &&
notify(messageCall, email);
```

The precondition allows `Player Response Process` to verify if `Advice Service` has been behaving correctly, that is, if its average response time is less than the predefined temporal threshold. The average is calculated on the last 20 responses of the default `Advice Service`. The recovery is composed of one single strategy and calls `Response Time Proxy Service`, which in turn invokes the best `Advice Service` (in terms of response time) among those being monitored.

We also define another simple supervision rule (from now on called *Store*):

```
postcondition:
store $response_time = $Resp_time;
```

to add the post-condition that is in charge of storing the response time of the default `Advice Service`. These values are then used in the evaluation of the precondition presented above. The user is informed of the call of the `Response Time Proxy Service` by means of a proper email message (`notify(messageCall, email)`).

The player can move over many countries, in which different languages are spoken. `Game Process` tries to retrieve questions from `Quiz Service`s in a language each player can understand. This can be checked through a dedicated rule that exploits `Language Verifier Service`. The rule (from now on called *Quiz*) applied to `Quiz Service` invocations is:

```
postcondition:
let $question = $internalData/quiz_service/question;
returnString(LanguageVerifierService_WSDL,
   "getLanguage", input + $question + input,output)==
   $internalData/player[codepoint-equal
   (id/text(),"playerID")]/favouriteLanguage;

recovery strategy:
retry(1) ||
rebind( QuizBackup_WSDL) && notify(messageRebind, email)
|| halt() && notify(email, messageHalt)
```

Notice that `Language Verifier Service` takes as input the `$question` alias, which is a string that represents the question returned by `Quiz Service`. If its language is different from the player's, the system tries to recall the same `Quiz Service` to verify if the problem is transient. If the service keeps using the wrong language, the self-healing infrastructure invokes another quiz service (`Quiz Backup Service`), notifies the rebinding to the player, and monitors the new interaction. If the monitoring phase triggers some problems, the system ignores them and it simply notifies the event to the player.

As already said, players should always receive suggestions that are long enough, yet still not exceissevly long. The rule (from now on called *Length*) on `Advice Service` is:

```
postcondition:
let $advice = $internalData/advice_service/advice;
let $condition1 = ($advice).length() > 20;
let $condition2 = ($advice).length() < 70;
$condition1 && $condition2;

recovery strategy:
if(!$condition1){
    change_monitoring_rules(new_monitoring_expression(*),
      new_recovery_strategy, permanent) &&
    notify(messageChangeMonitoringRule, email) ||
    ignore() && notify(messageIgnore, email)
  }
  if(!$condition2){
    rebind() && notify(messageRebind, email) ||
      call(SummarizerService_WSDL) &&
       notify(messageCall, email)
  }

(*) let $advice = returnString(AdviceService_WSDL,
        'getAdvice', input, output);
    let $condition1 = ($advice).lenght() > 10;
    let $condition2 = ($advice).lenght() < 70;
    $condition1 && $condition2;
```

In this rule we make extensive use of aliases to be less verbose, and to reuse conditions already evaluated: `$advice` is the string that keeps track of the suggestion returned by the invoked `Advice Service`, `$condition1` is true if the length of the suggestion is greater than 20 characters, and `$condition2` is true if the length of the advice is less than 70 characters. The recovery strategy fires if at least one of the two conditions is not verified. It is composed of two mutually exclusive clauses, respectively associated to the truth value of `$condition1` or `$condition2`. The recovery strategy can be adapted depending on the entity of the infraction. In fact if `$condition1` is not verified, recovery tries to relax the monitoring parameters by decreasing the minimum length of accepted suggestions to ten characters. The monitoring parameter, in this case, is persistent and thus the rule is modified permanently, but parameters can also be "BPEL instance" and thus any modification only applies to the particular instance. If the rule is still not verified, the system ignores the problem and notifies the player.

If `condition2` is not verified, the system first tries to rebind to another advice service (`Advice Backup`) and, if the monitoring rule is still not verified, the self-healing infrastructure calls (`Summarizer Service`) to reduce the length of the advice.

The last rule verifies that all players have more than zero points; if it is not the case, these players must be eliminated. The recovery strategy is performed with a `callback` and then if all players have less than zero points, the application is halted. We can easily see that `condition4` is stricter than `condition3` and thus it is defined first in the recovery strategy, since their prioritization is implicitly given by the order in which conditions are specified. Therefore, the recovery strategies associated with the infringement of `$condition4` are performed before those associated with the infringement of the other condition.

Notice that lower priority recovery might never be performed since their execution implicitly depend on the success of higher-priority ones. For example, lower priority strategies can become useless if a higher priority strategy halts the process, or if it unexpectedly fixes more than one infringement. However, the chosen solution based on a simple and flexible language to compose elements leaves such issues up to the designer of the supervision rule.

This rule (from now on called *Point*) has to be verified during the progress of the entire game, this is why it must be defined as invariant.

```
invariant:
let $condition3 =
   exists($p1 in $internalData/player, $p1/points <= 0);
let $condition4 =
   forall($p2 in $internalData/player, $p2/points <= 0);
$condition3 || $condition4;

recovery actions:
if($condition4) {
    halt() && notify(messageHalt, email);
}
  if($condition3){
    process_callback(event_name, params);
}
```

5. DYNAMO

Our approach is based on the integration of our aspect-based extension of ActiveBPEL (an open source and Java-based BPEL engine [1]) and *Dynamo* (i.e., our supervision framework)[3, 5]. Figure 2 presents the architecture of the self-healing framework and also gives an overview of the various technologies used to implement its different parts. The framework consists of two main components: (1) the

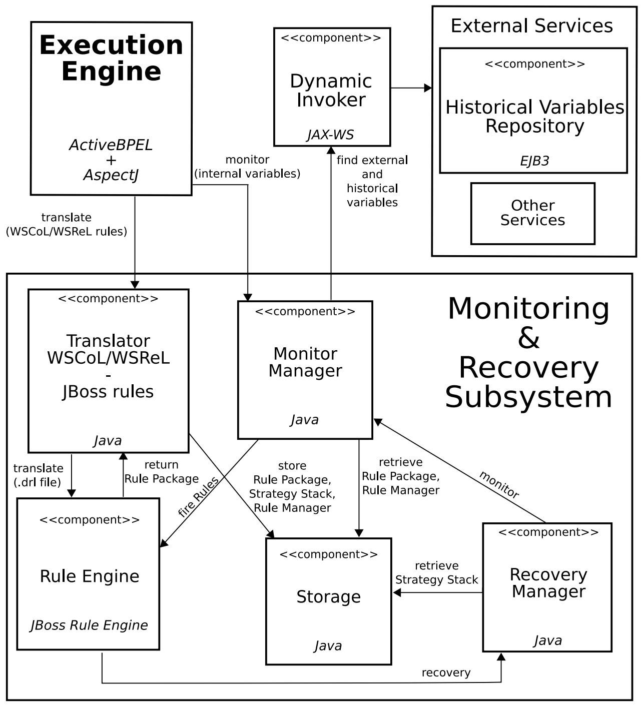

Figure 2: Self-healing framework.

extended BPEL engine (inspired by [3, 5], and based on ActiveBPEL and AspectJ [9]), and (2) the monitoring and recovery subsystem (based on JBoss rules technology [16]). The architecture also includes a number of utility components, like the *Dynamic Invoker* and the *Historical Variables Repository*. In this paper, we mainly concentrate on the JBoss-based implementation of the monitoring and recovery subsystem. As for the engine, we modified ActiveBPEL using AspectJ to define pointcuts responsible for activating monitoring and recovery strategies. Since we are only interested in monitoring the interactions occurring between the process and the outside world, we have decided to intercept the process only when it performs `invoke`, `receive`, and `pick` activities. As for Dynamo, the main idea behind our implementation is that supervision rules are translated to produce Drools v3.0 rules. The actual self-healing capabilities are therefore implemented on top of the JBoss Rule Engine. Moreover, the engine is in charge of verifying the monitoring expressions (pre-, post-conditions and invariants), and of activating the corresponding recovery strategies. Recall that a JBoss production rule is structured following the ECA (*Event - Condition - Action*) rule paradigm, and that it can be generalized as: when $<$ *condition* $>$ then $<$ *action* $>$, meaning that the *action* is executed if the *condition* is verified.

The elements of interest of the architecture of Figure 2 are:

- The Translator, which translates WSCoL and WSReL statements into the language required by the rule engine. In particular, it generates a (1) JBoss rule file, which contains the Drools rules corresponding to the supervision rules, (2) a *Rule Manager* (object) to manage the execution of the corresponding JBoss rules, and (3) a set of *Strategy Stacks*, one for each strategy step in a recovery strategy written in WSReL.
- The Monitor Manager, which is invoked when a rule in the BPEL process has to be verified. It manages the rule execution, controlling the order in which rules are verified by the rule engine. It also retrieves the data needed for the evaluation of the rules: internal variable are sent by the BPEL process and external and historical variables are retrieved respectively from external services and the *Historical Variable Repository*.
- The JBoss Rule Engine (JBRE), which, at compilation time, checks and compiles the JBoss rule file generated by the *Translator*, and, at run time, verifies the assertions and triggers and executes recovery strategies.
- The Recovery Manager, which executes the defined recovery strategies and, if needed, asks the *Monitor Manager* to perform extra monitoring steps.
- Storage, which is the database in which we store the rules, after they have been compiled by the JBRE, and the objects (Rule Managers and Strategy Stacks) needed for executing these rules. This component is implemented using *MySQL DB technology* [19].
- External Services, which are the external services invoked by the *Monitor Manager* to retrieve the necessary data to evaluate assertions. Among the esternal services we also include the *Historical Variables Repository*, which is deployed as a web service, adopting JBoss [15] and EJB3 [6] technologies.

Users start designing the self-healing capabilities of their BPEL process by specifying supervision rules. These rules are translated by the Translator into a set of JBoss rules that are then compiled by JBRE to produce a rule package stored in the Storage. Since rule compilation is a time-consuming operation, it is usually performed offline, before starting the execution of the process. In case a rule needs to be modified at execution time, only that rule wil be recompiled by the system. The Translator also produces and stores other objects that are needed to manage the execution of the rules.

As soon as the execution of the BPEL process reaches one of the pointcuts added to ActiveBPEL, Dynamo checks whether there is a supervision rule associated with the activity and whether it should be verified given its meta-data (i.e., priority and the other elements). If this is the case, the Monitor Manager first retrieves the compiled JBoss rules (from Storage) and adds them to the JBRE's Rule Base. Then, it proceeds to obtain all the required data from the process state and from external services. Internal variables are extracted using XMLBeans technology [2], while external variables (and historical variables) are obtained using the Dynamic Invoker (a component capable of invoking any external web service). In the case of historical variables, the data is obtained from the Repository of Historical Variables. In our implementation the repository is deployed in JBoss (using an underlying MySQL database), and historical variables are stored as EJB3 objects. Once data collection is completed, the JBRE can verify the monitoring expression.

If the expression is not verified, the JBRE activates the appropriate recovery strategy, inserted in the <*then*> clause of the JBoss rule. In particular, the activation of a recovery strategy consists in retrieving the *Strategy Stack* that corresponds to the recovery strategy from the Storage, and in executing the strategy steps by following the order in which they are inserted in the stack. At the end of any strategy step, a new monitoring phase could be necessary.

To better understand how the supervision rules are translated into the JBoss rule language, we need to briefly describe how the JBossRules engine works. The engine basically contains three state spaces: (1) a space containing all the rules that the system should consider for activation (*Rule Base*), (2) a space, called *Working Memory*, that contains the facts that might cause rules to activate, and (3) a space, called *Agenda*, that contains the rules that have been activated and that are waiting to be executed. To make things simple, each time a fact (represented by a JavaBean) is asserted (or added) to the Working Memory, the list of defined rules is searched. If there are no rules with conditions that evaluate to true, then their action parts are added to the Agenda where they wait to be fired. The content of the Agenda progressively grows as new facts are asserted to the Working Memory, and only when they are ordered to fire, they are actually executed (in a LIFO manner). Once the Agenda starts executing its rules, it tries to continue until there are no rules left.

5.1 Translation process

The key points to consider when translating WSCoL and WSReL into JBoss rules is that they will need to be evaluated in such a way as to mimic the semantics of defined supervision rules, and the mapping between BPEL variables and the data asserted in the Working Memory.

For each supervision rule, the compilation system creates one *Rule Manager*, responsible for managing the execution of the corresponding JBoss rule. At run time, before checking the rule, the *Rule Manager* is responsible for asserting the data needed to evaluate the rules into the Working Memory under the form of *XmlWrapper* objects, that is Java Bean objects that contain XML content. This implies that each object can contain more than one value, as long as each value is of a simple XSD type: string, boolean, or double. The variable type is interpreted at compile time depending on the context (i.e., the rule) in which it is referenced, and the Translator generates a JBoss rule in which the variable is interpreted according to the inferred type. If the system cannot assign a type at compile time, it creates more than one JBoss rule, to cover all the possible types that the variable could assume. At runtime the system finds the correct type and only fires the rule that gives the right interpretation.

A WSCoL expression with no aggregate operators is translated into a single JBoss rule. The agenda group of this rule is equal to the id of the Rule Manager that is associated with the WSCoL expression. This manger, before executing the rule, and after preparing the appropriate XMLWrapper objects, gives the focus to the correct agenda group to ensure that only the rules associated with that specific manager can be executed. For example, let us consider the following WSCoL rule:

```
$internalData/player/points > 0;
```

It has an internal variable, and no aggregate operators; therefore, the compilation system generates one JBoss rule and one Rule Manager —whose id is `manager_id`. The JBoss rule generated by the system is:

```
rule "rule_name"
agenda-group "manager_id"
when
   uuid : XmlWrapper(id == "uuid") &&
   eval(uuid.getDoubleValue() > 0);
then
end
```

`uuid` is a variable that univocally refers to the internal variable adopted in the WSCoL expression; and is an XML-Wrapper that has been asserted into the Working Memory. The condition that must be evaluated is encapsulated within the `eval` clause, in which we are allowed to insert pure Java code. It is important to highlight that the variable type is inferred at compilation time and that it is a double (see `uuid.getDoubleValue()`).

Since WSCoL provides aggregate operators, like `forall`, `avg`, `exists`, `sum`, etc., we can have WSCoL expressions that are translated into more than one JBoss rule. Each JBoss rule corresponds to a fragment of the original WSCoL expression. In this case, the Translator also generates more than one manager. For example, *Forall/Exists Managers* are in charge of evaluating the conditions inserted in a WSCoL forall/exists expression for all the values that the asserted WSCoL variable can assume until the quantified condition is either verified or not. For the evaluation of a quantified condition, we generate two mutually exclusive rules: one is verified if the forall/exists condition is true, and the other when it is false. This idea is exemplified through the following example:

```
forall($p2 in $internalData/player, $p2/points <= 0);
```

For this expression, the Translator generates one Rule Manager, one forall manager, and three JBoss rules: Rule 1 and Rule 2 are needed for the evaluation of the forall condition ($p2/points $\leq$ 0) and are mutually exclusive. Rule 0 is associated with the Rule Manager of the entire WSCoL expression.

```
rule "0"
agenda-group "manager_id"
when
   forall_uuid : ForallManager(id == "forall_uuid") &&
   eval( forall_uuid.getValue());
then
end

rule "1’"
agenda-group "forall_uuid"
when
   forall_uuid : ForallManager(id == "forall_uuid") &&
   uuid : XmlWrapper(id == "uuid") &&
   eval(uuid.getDoubleValue <= 0)
then
   forall_uuid.execute();
end

rule "2"
agenda-group "forall_uuid"
when
   forall_uuid : ForallManager(id == "forall_uuid") &&
   uuid : XmlWrapper(id == "uuid") &&
   eval(!(uuid.getDoubleValue <= 0))
then
end
```

The overall Rule Manager, before firing rule 0, triggers the rules associated with the forall operator within the WSCoL expression (this is achieved using the Forall Manager).

These rules have the same agenda group (whose id is `forall_uuid`), but different from that of the Rule Manager. If Rule 1 is verified, the *forall* manager can evaluate the forall condition for the next value of the variable, until there are no more values that the variable can assume. If there are no more values the forall manager sets its own validity value to true, and asserts it in the Working Memory. On the other hand, as soon as Rule 2 is verified the evaluation of the forall condition is blocked and the value of the forall manager is immediately set to false. Whatever the case, as soon as the Forall Manager itself is asserted to the Working Memory, the Rule Manager fires Rule 0.

Moving on to how we deal with WSReL strategies, we need to consider concepts such a a JBoss rule's "salience" or "activation-group". A salience is an integer value that gives a rule a certain priority (higher priority rules are executed before lower priority ones, while rules with the same priority are executed in a LIFO manner). An activation-group is a string that identifies a group of rules that must be executed in mutual exclusion, that is, only one rule out of the set can be executed. WSReL `if-then-else` statements translate into rules that are tagged with the same activation-group, meaning they are executed as explained above, while simple sequences of `if-then` statements translate into rules that have different salience values, giving them a uniquely identified execution order. The definition of these rules, along with an appropriate population of the Working Memory (in fact, the order in which facts are asserted is important), gives us the activation order we wanted.

As for the atomic actions, and how we understand if they are successful or not, the JBRE must first retrieve and assert all the data it needs; part of them may have already been asserted in the Working memory, while new data are retrieved and added. When monitoring needs to be re-evaluated, the Agenda is completely cleaned (i.e., for the parts regarding the given Agenda-group), and the Working Memory is updated to reflect the changes in monitoring data and monitoring results. After monitoring is completed, the Agenda and Working Memory are both completely cleaned. For example, if we think of the supervision rule that checks the length of received advices, the corresponding JBoss rules are:

```
rule "0"
agenda-group "manager_id"
when
   condition1 : XmlWrapper(id == "condition1") &&
   condition2 : XmlWrapper(id == "condition2") &&
   eval( condition1.getBooleanValue() &&
       condition2.getBooleanValue());
then
   fireRecoveryStrategy();
end

rule "Strategy_1 "
   salience 4;
   activation-group "cur_condition"
   agenda-group "length-postcondition"
when
   condition1 : XmlWrapper(id == "condition1") &&
   eval(condition1.getBooleanValue())
then
   recov_strategy_1();
end

rule "Strategy_2 "
   salience 4;
   activation-group "cur_condition"
   agenda-group "lenght-postcondition"
when
   condition2 : XmlWrapper(id == ''condition2'') &&
   eval(condition2.getBooleanValue())
then
   recov_strategy_2();
end
```

Rule 0 determines if a recovery strategy has to fire. Method `fireRecoveryStrategy` gives the focus to the Agenda group `"length-postcondition"`, that is, to the Agenda Group of the rules that represent the recovery strategy (`Strategy_1` and `Strategy_2`). Each `recov_strategy_X()` is a method that calls the implementation of the corresponding recovery strategy, defined for the violation of `condition1` or `condition2`.

6. PERFORMANCE EVALUATION

To evaluate the performance hit introduced by synchronous monitoring, we have gathered information regarding how much time it takes our prototype to translate and execute WSCoL rules. This evaluation was performed (over 1000 measurements) on the rules presented in Section 4.

Translation time. In evaluating rule translation times, we distinguish between the time it takes the JBoss Rule Engine to generate a rule package from a `.drl` file, and the total translation time, which also includes the translation from WSCoL to the `.drl` file. Table 1 shows the average, median and variance of the translation time. Column *%RE* shows how much of the time is due to JBoss' own translation algorithm. From these results we can see that rules containing aggregate operators or universal/existential quantifiers are longer to translate. Moreover, most of the translation time is in fact due to JBoss' own translation algorithm. Consider, for example, the *%RE* value for rule `store`, which is not translated into any JBoss rule.

	Average [s]	*Median [s]*	*Variance*	*%RE*
Store	0.004 s	0.002 s	$\approxeq 0$	27.05%
Average	0.120 s	0.089 s	0.002	95.31%
Length	0.092 s	0.075 s	0.002	89.98%
Quiz	0.091 s	0.072 s	0.002	92.71%
Point	0.209 s	0.219 s	0.006	93.35%

Table 1: Translation times.

Fortunately, translation is achieved off-line, during deployment. This means that we need not consider them when talking about pure run-time performance. At run-time all we need to consider is the amount of time it takes to extract already compiled rules from our MySQL database, an operation which is considerably faster.

Execution time. We have also collected performance results for execution time, distinguishing the time spent executing the rule, and the time spent collecting data from external variables and/or historical variables. Table 2 shoes the average, median, and variance of the execution times. Column *%EV* shows how much of the time is spent collecting external/historical data. A great deal of the execution time is dedicated to collecting data from external services, as we can see in rules *Store, Average,* and *Quiz*. Obviously, these execution times depend on the response times of the external services being considered, which can lead to higher variance in our results.

	Average[s]	*Median[s]*	*Variance*	*%EV*
Store	0.100 s	0.079 s	0.004	85.01%
Average	0.245 s	0.237 s	0.005	29.64%
Length	0.049 s	0.039 s	0.001	–
Quiz	0.205 s	0.133 s	0.042	72.82%
Point	0.118 s	0.106 s	0.002	–

Table 2: Execution times.

The performance hit, therefore, heavily depends on the amount of monitoring being achieved, and on the load conditions of the network. This explains why rule *Average* is the slowest to check. An aggregate operator needs to check its sub-rule as many times as the cardinality of the principal variable. Existential and universal quantifiers are treated similarly. Even though they stop execution as soon as possible, their worst case scenario is still the cardinality of the principal variable. Moreover, in this case rule *Average* also has to interact with external services to gather data.

In conclusion, the addition of monitoring capabilities only add milliseconds to the execution of a single BPEL invoke activity, which is encouraging.

7. RELATED WORK

The idea of using a rule engine to enforce self-healing policies on top of complex worklows is not new, but we think that our solution is more flexible and user-oriented than other solutions. For example, a complementary approach to our work is SHIWS (Self Healing Integrator for Web Services) [7] that exploits a self-adaptive approach, based on a mechanism for revealing possible runtime mismatches between requested and provided services, and for dynamically adapting the client application accordingly. The authors are interested in revealing mismatches between different implementations of the same contract and in finding a set of adaptation strategies for the classes of possible mismatches. The main difference with respect to our approach is that we aim to increase the self-healing capabilities of deployed BPEL processes.

Another approach [11] proposes the combination of concepts from autonomic computing and Web services for achieving self healing capabilities. The authors aim to categorize Web services in two categories: functional Web services and autonomic Web services. A functional Web service can behave autonomically by discovering and using other autonomic Web services over the Internet without implementing autonomic attributes. Autonomic web services are responsible for providing the self-healing autonomic attributes to enable functional Web services to discover, diagnose, and react to unexpected events that would lead to malfunctions. This approach uses a PDM (Problem Determination Mechanism) to discover the root cause of the problem, and then recommends some actions to correct identified problems. It also uses a rule engine to determine the actions that can be taken according to defined system policies contained in a database (named Symptom database). The differenc is that we embed self-healing capabilities within BPEL process through weaving of supervision rules by means of AOP techniques, while in this approach a system inherits self-healing capabilities by invoking special-purpose external services.

Rule engines are also widely exploited in recovery management. DIOS++ [18] is a framework for rule-based autonomic adaptation and for controlling distributed sensor-monitored scientific applications. RuleBAM [17] is another framework that uses Business Activity Management (BAM) polices to define system requirements and to automatically produce executable business rules that implement recovery. These approaches however are not specifically tailored towards BPEL processes. They also do not provide specific mechanisms for complex strategies, but only atomic actions.

Another approach [8] exploits a middleware framework called Robust Execution Layer (REL) that acts as a transparent, configurable add-on to any BPEL execution engine to support the self-healing execution of business processes that are managed by the engine. The basic idea of REL is to provide handling capabilities for low-level communication faults in the interaction of a business process engine with external Web service providers.

Another approch focused on users' needs is explained in [20]. This paper points out the research activities needed to perform the design and the execution of reliable scientific BPEL workflows. The authors suggest adopting a Failure investigation service. The various components involved in scientific workflows are monitored through autonomic recovery strategies which take action to prevent the various parts of a workflows from terminating abnormally. Users are notified about any incidents that occured and any actions that were taken during execution through reports.

The approach presented in [12] suggests the introduction of programming environments, such as exception handling and atomicity, in business workflows. The adopted recovery strategies are based on backward recovery: in the case of an error, an application or parts of it are rolled back to a previous consistent state. All necessary steps for undoing work are performed by the runtime system on the base of logged information.

8. CONCLUSIONS AND FUTURE WORK

We have presented Dynamo, a flexible and programmable solution for self-healing BPEL processes. Through separation of concerns, the use of AOP-techniques, and a rule engine, we have been able to provide a solution that is dynamic and capable of leveraging the intrinsic characteristics of the "open-world" assumption.

We have used a complex case study to exemplify the use of the WSCoL and WSReL languages for ensuring self-healing processes. Moreover, we have presented an entirely new implementation of Dynamo, based on the use of a rule engine, and pointed out its main advantages.

Future work will concentrate on new kinds of recovery actions, and take advantage of the knowldege gathered so far. In particular, we will study intra-process recovery techniques such as rollback and dynamic reconfiguration.

9. REFERENCES

[1] Active Endpoints. ActiveBPEL. `http://www.activebpel.org/docs/architecture.html`.

[2] Apache. XMLBeans. `http://xmlbeans.apache.org/`.

[3] L. Baresi and S. Guinea. Dynamo: Dynamic Monitoring of WS-BPEL Processes. In *5th International Conference on Service Oriented Computing*, pages 478–483, 2005.

[4] L. Baresi and S. Guinea. Towards Dynamic Monitoring of WS-BPEL Processes. In *5th International Conference on Service Oriented Computing*, pages 269–282, 2005.

[5] L. Baresi and S. Guinea. Dynamo and Self-Healing BPEL Compositions. In *29th International Conference on Software Engineering (ICSE'07 Companion)*, pages 69–70. IEEE Computer Society, 2007.

[6] B. Burke and R. Monson-Haefel. *Enterprise JavaBeans 3.0 (5th Edition)*. O'Reilly Media, Inc., May 2006.

[7] G. Denaro, M. Pezzè, and D. Tosi. SHIWS: a Self-Healing Integrator for Web Services. In *29th International Conference on Software Engineering (ICSE'07 Companion)*, 2007.

[8] T. Friesel, J. P. Muller, and B. Freisleben. Self-healing Execution of Business Processes Based on a Peer-to-Peer Service Architecture. In *Systems Aspects in Organic and Pervasive Computing - ARCS 2005*, volume Volume 3432/2005 of *Lecture Notes in Computer Science*, pages 108–123. Springer Berlin / Heidelberg, 2005.

[9] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold. An Overview of AspectJ. In *ECOOP 2001 - Object-Oriented Programming: 15th European Conference, Budapest, Hungary, June 18-22, 2001, Proceedings*, volume Volume 2072/2001, page 327. Springer Berlin / Heidelberg, 2001.

[10] S. Guinea. *Dynamo: a Framework for the Supervision of Web Service Compositions*. PhD thesis, Politecnico di Milano - Dipartimento di Elettronica e Informazione, 2007.

[11] S. A. Gurguis and A. Zeid. Towards Autonomic Web services: Achieving Self-Healing using Web services. In *DEAS '05: Proceedings of the 2005 workshop on Design and evolution of autonomic application software*, pages 1–5, New York, NY, USA, 2005. ACM Press.

[12] C. Hagen and G. Alonso. Exception Handling in Workflow Management Systems. *IEEE Trans. Softw. Eng.*, 26(10):943–958, 2000.

[13] IBM, Autonomic Computing Initiative. Autonomic Computing. `http://www-03.ibm.com/servers/autonomic/`.

[14] IBM, BEA Systems, Microsoft, SAP AG, Siebel Systems. Business Process Execution Language for Web Services version 1.1. `http://www.ibm.com/developerworks/library/specification/ws-bpel/`.

[15] JBoss. `http://labs.jboss.com/`.

[16] JBoss. JBoss Rules. `http://labs.jboss.com/jbossrules/docs`.

[17] J. Jeng, D. Flaxer, and S. Kapoor. RuleBAM: A Rule-Based Framework for Business Activity Management. In *IEEE SCC*, pages 262–270. IEEE Computer Society, 2004.

[18] H. Liu and M. Parashar. DIOS++: A Framework for Rule-Based Autonomic Management of Distributed Scientific Applications. In *Euro-Par*, pages 66–73, 2003.

[19] MySQL. `http://www.mysql.com/`.

[20] B. Wassermann and W. Emmerich. Reliable Scientific Service Compositions. In G. Feuerlicht and C. Zirpins, editors, *Proc. of 2nd Intl. Workshop on Engineering Service-Oriented Applications: Design and Composition, WESOA'06*. Springer Verlag, December 2006.