Self-supervising BPEL Processes

Luciano Baresi and Sam Guinea
Politecnico di Milano – Dipartimento di Elettronica e Informazione
via Golgi 42 - I-20133, Milano, Italy
{baresi, guinea}@elet.polimi.it

***Abstract*— Service compositions suffer changes in their partner services. Even if the composition does not change, its behavior may evolve over time and become incorrect. Such changes cannot be fully foreseen through pre-release validation, but impose a shift in the quality assessment activities. Provided functionality and quality of service must be continuously probed while the application executes, and the application itself must be able to take corrective actions to preserve its dependability and robustness.**

We propose the idea of *self-supervising* BPEL processes, that is, special-purpose compositions that assess their behavior and react through user-defined rules. Supervision consists of monitoring and recovery. The former checks the system's execution to see whether everything is proceeding as planned, while the latter attempts to fix any anomalies. The article introduces two languages for defining monitoring and recovery and explains how to use them to enrich BPEL processes with self-supervision capabilities. Supervision is treated as a cross-cutting concern that is only blended at runtime, allowing different stakeholders to adopt different strategies with no impact on the actual business logic. The paper also presents a supervision-aware run-time framework for executing the enriched processes, and briefly discusses the results of in-lab experiments and of a first evaluation with industrial partners.

***Index Terms*— D.2.4.a [Software Engineering]:Software/Program Verification - Assertion checkers, assertion languages, performance; D.2.2.c [Software Engineering]: Design Tools and Techniques - Distributed/Internet based software engineering tools and techniques**

I. Introduction

The Oasis consortium defines Service Oriented Architectures (SOAs) as [1]: *"A paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. It provides a uniform means to offer, discover, interact with and use capabilities to produce desired effects consistent with measurable preconditions and expectations."* The importance of this definition is twofold: it emphasizes the intrinsic distributed ownership of these applications, and it highlights the need for "desired effects consistent with measurable preconditions and expectations".

Many researchers have studied how to provide and regulate desired effects [2], [3], [4] in software systems, but the peculiarities of service-oriented applications make it difficult to understand how they can be ensured through the entire life-cycle. Services usually only provide a syntactical description of their interfaces, loose coupling and distributed ownership allow individual services to evolve separately, and late binding techniques support their run-time selection. All these characteristics open the door to potentially unexpected (and often undesired) changes in functionality and QoS while the application is in operation. This ephemerality requires the continuous assessment of system properties, and thus conventional testing approaches must be complemented with run-time validation techniques [5].

This article contributes to this last aspect and proposes a technique, along with supporting tools, for deploying robust and dependable Web services[1] [6] compositions in the form of BPEL (Business Process Execution Language, [7]) processes. BPEL only supports the workflow-based composition (orchestration) of external partner services: a centralized entity —usually called *engine*— is in charge of the synchronization of the different services, which provide the main computational capabilities. This is why the proposal privileges the process-side supervision of the interactions between the process and its partner services. The points of the process we are interested in are those in which it interacts with the outside world. This way we can check whether partner services, which may have evolved independently, are still appropriate for our needs. *Supervision rules* declaratively specify both the monitoring directives, which synchronously check predefined points of the process to see whether everything is proceeding as planned, and the reaction strategies, which activate some form of recovery when anomalies arise. Together, monitoring and recovery allow us to speak of *self-supervising* BPEL processes.

Monitoring directives are similar to conventional *design by contract* [2], but it is the client (and not the provider) that specifies its own pre- and post-conditions. For example, a method clearly states what it can offer and under what conditions. Web services do not usually provide such information, and this is why we decided to flip the perspective. Pre- and post-conditions must be interpreted as expectations before and after the process interacts with its partner services. Borrowing from well-known assertion languages (e.g., Anna [8] and JML [9]), monitoring directives are expressed in WSCoL (Web Service Constraint Language), which is our special-purpose language that mixes typical propositional logic constructs with XML-based technology to provide a level of abstraction BPEL designers are familiar with. *WSCoL* concentrates on both functional and non-functional properties, an is suitable to express general dependability properties such as safety (i.e., absence of catastrophic events), integrity (i.e., no improper state alterations), availability (i.e., readiness for service), and reliability (i.e., continuity of correct service).

Recovery strategies follow the typical ECA (event-condition-action) paradigm and are stated in our WSReL (Web Service Recovery Language). The event is the discovery of a run-time anomaly. This means recovery starts as soon as a monitoring assertion signals an error. The condition is once again expressed in WSCoL, and allows us to choose among alternative recovery options, which are defined by picking and mixing atomic recovery actions from an easily extensible library of predefined actions.

[1]Since our work concentrates on Web service technology, from now on, we will use the terms service and Web service as synonyms.

Actual recovery capabilities heavily depend on the services the process interacts with. Stateless partner services simplify the problem. Things become more complex when the process interacts with stateful or conversational services. The former are services that have persistent side-effects when called (e.g., business data are stored on a persistent database). This means that calling them more than once may not lead to repeated behavior, and that they must provide a special operation if we want to be able to undo their effects. The latter require that a special conversation protocol be respected. In these cases, we need a way to rollback the conversation itself. This is why the use of stateful and conversational services can lead to situations in which only partial recovery is possible.

BPEL itself would allow designers to mix *defensive programming* techniques [10] and fault, event, and compensation handlers with the actual business logic to embed supervision into the process, but this solution would be complex (due to the limited capabilities of the language), and inflexible (any change would require modifying the BPEL process and redeploying it). In contrast, our approach fosters *separation of concerns* since it maintains the actual business logic and supervision directives separate at design time. Designers define the process' business logic, without considering supervision; then they declaratively specify their supervision rules. At runtime, the two elements are intertwined by means of AOP (Aspect Oriented Programming [11]). The weaving allows us to differentiate the supervision directives we want to consider at runtime, based on *who* is running the process, *when* it is being run, and in collaboration with *what* partner services. This is achieved without modifying the process and/or re-deploying it.

This article presents a comprehensive treatment of our proposal for self-supervising BPEL processes. Some preliminary versions of the work were presented in [12], [13], but this article provides a uniform and comprehensive presentation of all the framework's capabilities, after extensive improvement. More specifically, we have extended the semantics of meta-level information, that is, of the information designers can use to clarify whether a supervision rule must be considered or ignored at runtime. We have reconsidered response times in WSCoL and implemented a new data collector for gathering them. We have reworked WSCoL and WSReL to improve their interplay. We have added backward recovery to WSReL, that is the capability to restore a previous point in the process' execution. Finally, we have conducted thorough evaluation of the approach, by investigating in-lab performance issues, and by using it with both real-world industrial partners and students.

The rest of the article is organized as follows. Section II provides a brief introduction to the BPEL language and to the case study we use throughout the article. Section III describes our supervision approach, while Sections IV and V introduce WSCoL and WSReL, respectively. Section VI concludes the presentation of supervision rules with some significant examples. Section VII illustrates the prototype execution environment and the evaluation we carried out, and discusses the important lessons learned. Section VIII surveys the state of the art and Section IX concludes the article.

II. Running example

This section introduces the main elements of BPEL, to make the article self-contained, the running example used throughout the article, and some informal supervision requirements.

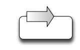
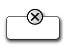
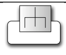
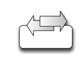
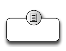
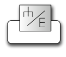
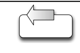
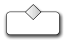
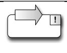
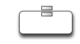
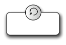
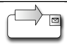
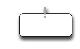
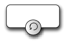
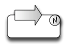
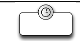
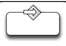
Activity	Shape	Activity	Shape	Activity	Shape
receive		terminate		flow	
invoke		sequence		forEach	
reply		if		fault handler	
assign		while		event handler	
throw		repeatUntil		comp. handler	
wait		pick			

Fig. 1. Graphical notation for BPEL.

A. BPEL in a nutshell

BPEL 2.0 [7], Business Process Execution Language (for Web Services), is a high-level XML-based language for the definition and execution of business processes by means of Web service-based workflows.

The definition of a process contains a set of global variables and the workflow logic expressed as a composition of *activities*, where implicit or explicit *scopes* help define variables and activities at different visibility levels. BPEL does not come with a standard graphical representation, but Figure 1 introduces the graphical symbols used in this article to render its basic and structured activities, and its fault, event, and compensation handlers.

Activities include primitives for communicating with other services (*receive*, *invoke*, *reply*), for executing assignments (*assign*), for signaling faults (*throw*), for pausing (*wait*), and for stopping the execution of the process (*terminate*). All the primitives that communicate with the outside world use BPEL's supporting notion of *partnerlink* to describe with whom they want to communicate. Late binding is supported by using *assign* activities to change the endpoints associated with the partnerlinks at runtime. The activities *sequence*, *while*, *repeatUntil*, and *if* provide standard control structures to order activities, and define loops and branches. The *pick* is peculiar to the domain of concurrent and distributed systems, and waits either for the first message (out of several incoming ones) to occur or for a time-out alarm to go off, to execute the activities associated with such an event.

The *flow* supports the concurrent execution of activities. Synchronization among the activities of a *flow* may be expressed using the supporting notion of *link*s; a *link* can have a guard called *transitionCondition*. Since an activity can be the target of more than one *link*, it may define a *joinCondition* for evaluating the *transitionCondition* of each incoming link. By default, if the *joinCondition* of an activity evaluates to false, a fault is generated. Alternatively, BPEL supports *Dead Path Elimination* (DPE), to propagate a false condition rather than a fault over a path, thus disabling the activities along that path. *forEach* supports the concurrent or sequential execution of BPEL scopes: in the former case, the execution of its internal activities is serialized; in the latter case, they are executed as parallel flows.

Each scope (including the top-level one) may contain the definition of the following handlers:

- An *event handler* reacts to an event by executing — concurrently with the main activity of the scope— the activity specified in its body. In BPEL, there are two types of events: message events, associated with incoming messages, and alarms based on timers.
- A *fault handler* catches faults in the local scope. If a suitable fault handler is not defined, the fault is propagated to the enclosing scope.
- A *compensation handler* restores the effects of a previously completed transaction and is initiated programmatically by using a *compensate* activity.

B. Tele-radiology case study

The running example considers tele-radiology as a means to overcome the lack of specialized radiologists in certain geographical areas[2]. Instead of moving patients to private clinics every time bio-imaging is needed, technicians perform the imaging, and then send them to a remote specialized center for analysis (`TMC`). The result is a general improvement both in the average waiting time for patients and in costs.

Figure 2 shows the BPEL process that manages the interactions among patients, hospitals, and the telecare system. It exemplifies most of BPEL's modeling features, making it is a good compromise between generality and complexity. Its merits go beyond the particular problem it addresses and can be considered a good example of an average BPEL process.

A blocking *pick* allows the process to perform three main activities. In the first branch (called `makeReservation`), a patient can request a visit for a magnetic resonance. The process receives the patient's ID and checks to see if the request has also been advanced by a recognized doctor (*invoke* activity `checkMedicalRequest`). If this is not the case the process issues a *reply* with a negative answer (*reply* activity `negativeResult`). If the request can proceed, the process obtains a set of time slots (*invoke* activity `getCalendarDates`) and sends them to the patient so that he/she can choose one (*invoke* activity `sendDates`). When the patient responds (*receive* activity `receiveDate`), the process stores the reservation (*invoke* activity `storeDate`) and gives the patient a positive result (*reply* activity `positiveResult`).

In the second branch (called `performVisit`), a technician accesses the system to check whether the reservation is correct (*invoke* activity `checkReservation`). If it is not (e.g., the current time slot is not reserved for that patient) the process issues a negative answer and terminates (*reply* activity `negativeResult`). If the visit can proceed, the technician produces the magnetic resonance image, and passes it to the process (*receive* activity `receiveMRI`), which stores the image (*invoke* activity `storeMRI`) and replies positively (*reply* activity `positiveResult`).

In the third branch (called `getAnalyses`), a technician can proceed to forward magnetic resonance images to the telecare service. The process retrieves a batch of images from a storage component (*invoke* activity `getImages`), then sends them to the TMC (*invoke* activity `sendImagesToTMC`). The process waits for the analyses to become available through *event handler* `receiveAnalyses`. The analyses are checked to see if the specialists have uncovered any problems. If everything is fine, an analysis contain a "green code", if not a "red code". In the first case the patient is notified to go to the hospital and pick up the original scan (*invoke* activity `notifyPatient`). In the second case, the patient is notified and a digital copy is sent immediately to the his/her doctor (*invoke* activity `notifyPersonalDoctor`).

The process also provides a second *event handler* called `changeBinding`. This is used to dynamically modify the binding the process has with the telecare service. It receives the URI of the telecare service the process should use from that point on. This is achieved using *assign* activities that modify the telecare's endpoint references.

C. Supervision requirements

So far, we only considered the business logic, but this process needs careful supervision to turn it into a robust and dependable application. There is no single way to design and implement its supervision. The following supervision requirements are aimed at demonstrating what the proposed approach offers, and do not provide a complete supervision solution. For the sake of simplicity, from here on, we will refer to each requirement with a unique name.

CheckReservation requires that the reservation number, returned by *receive* activity `receiveDate`, be correctly encoded. If the code is wrong we may restore the process to before performing activity `getCalendarDates`, and switch to a backup appointment manager service. This way the process will refresh the possible dates by invoking activity `getCalendarDates` on the new service, and ask the user to choose a new one by re-invoking activity `sendDates`.

CheckCenter imposes that the reliability of method `sendImagesToTMC`, calculated over the last two hours, be at least 95%. Reliability is calculated as the number of times the method responded within 2 minutes, over the total number of invocations. If the reliability is too low the process must react by changing its binding to a backup service. However, we only change the telecenter's endpoint reference if it still points to the original telecenter service. If the endpoint has already been modified, for example through event handler `changeBinding`, the supervision rule is no longer relevant and we switch it off.

CheckResolution says that any image inserted in the system (through *receive* activity `receiveMRI`) must have a resolution that is between 800×600 and 1024×768 pixels. If this is not the case, we can decide among different strategies. In case the resolution is too high we can use an external service to lower the images' resolutions. If the resolution is not ideal, but not "too" low (within a 10% of our desired resolutions), we simply ignore the problem and let the process continue. If, on the other hand, the error is noticeable, there is nothing we can do, so we notify the head doctor and halt the process.

III. Our Supervision Approach

Our approach augments BPEL processes with self-supervising capabilities. This is achieved by defining appropriate *supervision rules*. Each rule must indicate the precise point in the process in which it is considered. This is done by specifying the rule's

[2]The example is loosely inspired by an important case study presented in the eHEALTHIMPACT project, supported by the European Commission Information Society and Media DG. We have adapted the case study by dropping most of the medical details. In no way do we presume it to have medical merit, but our revision was aimed at identifying a sufficiently complex case study to demonstrate the main features of our approach.

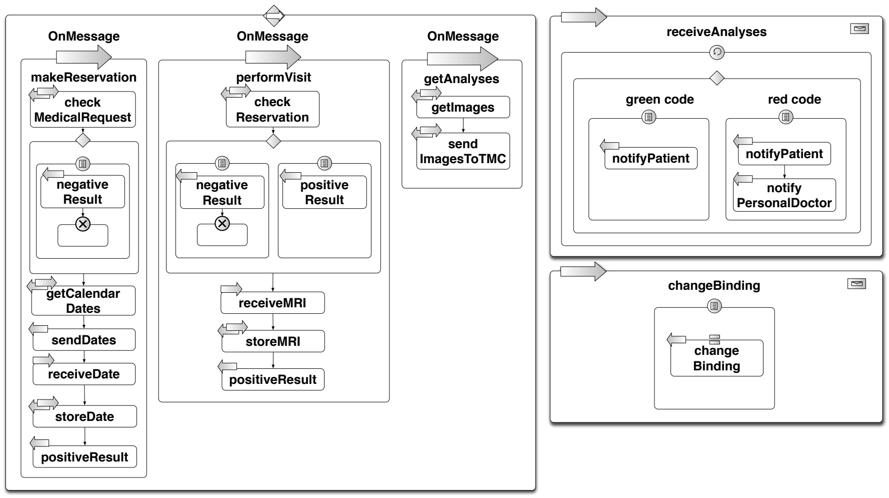


Fig. 2. The tele-radiology process.

`location`, that is an XPath expression that uniquely identifies a BPEL *invoke, receive, reply,* or *pick* activity within the process definition. This means that any BPEL activity that interacts with the outside world is a valid location. When defining the location we also specify if the rule is to be considered before or after the activity's execution (i.e., if it is a pre- or a post-condition).

Each rule also contains a set of `supervision parameters`. This meta-level information is used at runtime to decide whether a rule needs to be considered or not. The reason is that supervision necessarily introduces a performance overhead, and we want to be able to tailor the exact amount of supervision depending on the needs at hand without changing or redeploying the process.

Finally, a rule is made up of a monitoring expression, specified in WSCoL, and a set of alternative recovery strategies, specified in WSReL. Although these two key aspects cannot be treated entirely separately, since they influence each other, Section IV introduces WSCoL, while Section V presents WSReL and clarifies how the two work together.

A. Supervision Parameters

Supervision parameters allow the designer to tailor the degree of supervision that will be achieved by specifying when a rule can be "switched off". Our supervision parameters are *priority*, *validity*, *delay*, and *trusted providers*. All four are optional, and for each there is a default meaning.

Priority allows us to create layered sets of supervision rules by providing each rule with a numeric priority value. When a supervision rule is about to be checked, we compare its priority with a global process threshold value. All rules that have a priority equal to or less than the threshold are considered for supervision; those that do not are ignored. It is like having a knob that can be turned to increment or decrement the supervision activities being performed. If the priority is not given, it defaults to the lowest possible value, meaning the rule is always considered.

A *validity* defines a time window in which, if the process is run, the rule is checked. On the other hand, every time the process is run outside this window the rule is ignored. A validity parameter is defined by using two optional values: `from` and `to`. If both are specified, the time window has both a lower and an upper bound. If only the `from` value is specified, the supervision rule will be checked from that point in time on, and if only the `to` value is specified, the rule is checked up to that point. We also support a second interpretation of validity. In this case we use another keyword: `times` with a positive integer that states the maximum number of times a rule should be checked. If no validity window is given the rule is always checked.

Delay is supported through the use of keyword `wait for` followed by the definition of a temporal unit. After evaluating the rule once, the framework will wait for the delay to elapse before considering the rule again. Since we do not know exactly when a rule will be evaluated, this parameter specifies the minimum delay between subsequent evaluations. We also provide a second interpretation of delay. In this case we state the number of times we will wait for the rule to be considered before checking it again. If no delay value is given the rule is always checked.

Trusted providers is a list of service providers. If we interact with one of these providers, that interaction can go unsupervised. This is useful when we consider systems that adopt late-binding techniques. Again, if no trusted providers are defined the rule is verified for all providers.

IV. MONITORING

WSCoL (Web Service Constraint Language) is the assertion language we defined to specify what the process expects from partner services. It is evocative of assertion languages, such as ANNA (Annotated Ada [8]), and JML (Java Modeling Language [9]), but given the syntax of BPEL and Web services, WSCoL also takes inspiration from XML technology (e.g., XPath).

WSCoL is holistic. We do not limit ourselves to making assertions on the process' internals; we also consider aspects regarding the environment the process is run in. Our definition of environment is very broad, and comprises whatever data we can collect at runtime through external probes. For example, we can also predicate on data belonging to previous process executions. This is reflected in the three kinds of variables handled by WSCoL.

A. Variables

WSCoL variables can be either anonymous, and only available in the expression in which they are declared, or named locally within the rule in which they are defined. *Aliasing* (`let`) allows one to identify variables with simple names, and greatly simplifies expressions, which become less verbose. It allows a particular value to be collected only once, and be referenced any number of times in the rest of the supervision rule. Aliases can also be used to identify entire WSCoL assertions to allow for simpler definitions, identify analyses to be performed only once, and simplify recovery strategies.

The simplest variables managed by WSCoL are called *internal variables* and hold values that exist within the process in execution. They can be values passed by the process' client as input parameters, values received from the outside world through *receive*s, *pick*s, or *invoke*s, or values calculated by the process itself by using *assign* activities. Internal variables contain simple XSD values (i.e., String, Number, or Boolean), and should not be confused with BPEL internal variables, which typically match complex XSD types defined in WSDL interfaces. Indeed, a WSCoL internal variable provides a way of extracting simple XSD values from complex BPEL variables. This is done by specifying the name of a BPEL variable and an XPath expression. The result is either a single-valued variable or a container for multi-valued variables. For example, our running example suggests an internal variable for holding a reservation number:

```
let $res = $StoreReservationResponse/code
```

The expression defines alias `$res`: the BPEL variable `StoreReservationResponse`, introduced by the use of the `$` sign, contains the reservation code we want.

External Variables hold values that do not exist within the process and that must be obtained through the environment. This is achieved by interacting with any external probe that provides a WSDL interface. This choice eases the deployment of new probes and also fosters the re-use of third party entities.

An external variable is defined by means of the URI associated with the WSDL interface of the probe, the input message to be sent, and the XPath expression to be used to extract the value of interest from the message returned by the method. For example, in our case study we can assume an external variable to hold an image's resolution. This is not known by the process, nor can the process calculate it. What the process knows is where the image can be retrieved and how to store it in the system. The external variable is written as:

```
let $imageURL = $submitImageRequest/imageURL;
let $hRes =  return('WSDL_URI','<imageRes>'+
             $imageURL+'< /imageRes>',
             /getResResponse/horizontalRes)
```

The alias uses an internal variable (`$submitImageRequest/imageURL`) to build the message to be sent to the external probe. The XPath expression extracts the image's horizontal resolution from the message returned by the web method.

External probes open the door to quality of service metrics. For example, we often use a special probe in our post-conditions to obtain the response times of service invocations. Notice that, since we place the probe at the process' side, the response time is the sum of the service's execution time and the extra time due to the request and response messages transiting through the web. By definition, if a service does not respond it is impossible for us to calculate a response time (and thus evaluate the post-condition). Usually, this case is captured by the execution engine, which launches a special timeout exception[3] that is translated into a fault and propagated through the process. To avoid the engine taking over as the recovery manager, we catch the exception before it is propagated to the process, assume the post-condition is violated, and proceed directly to our own WSReL recovery strategies.

Historical Variables hold values related to previous process executions. WSCoL provides two functions for managing historical variables: `store` is used to store any kind of WSCoL variable to a persistent storage, and also implicitly define an alias, while `retrieve` is used to fetch a variable previously stored. For example, every time *invoke* activity `sendImagesToTMC` is executed, we could store its response time (a value obtained using an external variable):

```
store $rt = return(...);
```

`$rt` is a new name by which the value will be known and stored. When a variable is stored it is implicitly tied, through the engine, to the user that executed the process. The first time we execute a `store`, we create the alias and store its first value, while further executions simply add new values. The consistency and coherence of used names is up to the designer.

Function `retrieve` takes as input a variable name, but we can also supply an optional user identifier, and an optional time interval. The user identifier allows us to only retrieve the variables stored by a given user. The default is to fetch all the variables stored with that name, regardless of who stored them. The interval can be used in two ways: we can either indicate the maximum number of results that should be returned, or the maximum amount of time the function should consider when going back into the past to collect its values. In our example we can retrieve the values of variable `$rt` stored in our previous example.

```
retrieve('rt', null, 24h);
```

In this case we do not specify any particular user identifier, but use keyword `null` to specify that we are interested in all the values of `$rt` stored in the last 24 hours.

B. Constructs

The syntax for WSCoL assertions is defined as:

$$
\begin{array}{lcl}
\langle\mathsf{asrtn}\rangle & ::= & \neg\langle\mathsf{asrtn}\rangle \mid \langle\mathsf{asrtn}\rangle\&\&\langle\mathsf{asrtn}\rangle \mid \langle\mathsf{asrtn}\rangle||\langle\mathsf{asrtn}\rangle \mid \\
& & (\langle\mathsf{quant}\rangle\ \langle\mathsf{alias}\rangle\ \mathit{in}\ \langle\mathsf{values}\rangle,\ \langle\mathsf{asrt}\rangle) \mid \\
& & \langle\mathsf{term}\rangle\langle\mathsf{rop}\rangle\langle\mathsf{term}\rangle \\
\langle\mathsf{term}\rangle & ::= & \langle\mathsf{var}\rangle \mid \langle\mathsf{term}\rangle\langle\mathsf{aop}\rangle\langle\mathsf{term}\rangle \mid \langle\mathsf{const}\rangle \mid \\
& & \langle\mathsf{var}\rangle.\langle\mathsf{sfun}\rangle(\langle\mathsf{term}\rangle*) \mid \\
& & (\langle\mathsf{afun}\rangle\langle\mathsf{alias}\rangle\ \mathit{in}\ \langle\mathsf{var}\rangle,\ \langle\mathsf{term}\rangle) \\
\langle\mathsf{rop}\rangle & ::= & < \mid \leq \mid == \mid \geq \mid > \\
\langle\mathsf{quant}\rangle & ::= & \mathit{forall} \mid \mathit{exists} \mid \mathit{numOf} \\
\langle\mathsf{aop}\rangle & ::= & + \mid - \mid \times \mid \div \mid \% \\
\langle\mathsf{sfun}\rangle & ::= & \mathit{abs} \mid \mathit{replace} \mid \mathit{substring} \mid \ldots \\
\langle\mathsf{afun}\rangle & ::= & \mathit{sum} \mid \mathit{avg} \mid \mathit{min} \mid \mathit{max} \mid \mathit{product}
\end{array}
$$

where var is a variable, a variable alias, or a special purpose alias called `$instanceID` which returns the ID of the process currently being run, sfun are simple functions that mimic those commonly used in XPath, and afun are aggregate functions meant to be used with variables that have multiple values (containers). Boolean, relational (relop), and arithmetic operators (arop) follow their usual definitions.

[3] The length of this timeout can be configured.

Designers can use universal and existential quantifiers to express constraints over finite sets of values[4]. Their meanings are straightforward. When using a quantifier, the designer must define three parts. The alias names a variable that will be used as parameter in the upcoming assertion, values uses the syntax shown previously for variables to define the range of values that the alias can assume, and assertion defines the predicate we want to check.

V. Recovery

WSReL (Web Service Recovery Language) extends upon the legacy of WSCoL to provide a programmable, flexible, and extensible solution for both *local* and *backward* recovery. Local recovery tries to fix the anomaly in the current state of error, in a way that is similar to compensation. Indeed, once the corrective actions are performed, the system tries to continue its normal execution from the same state. Backward recovery, on the other hand, tries to restore the system to a previous state in which the anomaly was not present.

BPEL supplies compensation handlers, associated with scopes, to indicate that the activities within the scope are to be considered reversible in an application-defined way. There are however limitations, since a compensation handler only becomes active once its scope has completed successfully. It can be called either programmatically or from a handler associated with a further enclosing scope. Such limitations, together with the decision to associate compensation with scopes, denies a clear separation of concerns, causing business logic and recovery to be intertwined, and greatly complicates the definition of the recovery and of the process itself.

In the realm of database technologies [14], [15], and in more classical workflow-based systems [4] rollback is more common. Even though we have had distributed databases and workflow systems for quite some time, these systems have always lived in a world of well-defined rules and interactions. On the contrary, in BPEL we must cope with a new notion of distributed ownership. Different parts of the system are owned by different institutions, making it harder to perform true rollback.

WSReL provides a general solution by means of a series of atomic recovery actions, which are treated as building blocks, and language constructs to mix them to create more complex strategies. A WSReL expression is tightly related to the monitoring expression that allowed us to catch the run-time anomaly. From a monitoring expression given in WSCoL, we bring over any aliases we deem useful, whose values are collected during the monitoring phase. This means that recovery automatically knows and can use all the data that were collected during monitoring. Notice, however, that recovery can always define new aliases[5] if needed.

Currently, all recovery strategies are limited to instance validity. Designers can always modify and re-deploy their processes, but we do not provide any automatic solution for this step.

A. Atomic Recovery Actions

The atomic actions we provide can be organized in four main groups. The first group is made up of simple actions that do not modify how the process is executing. `ignore` allows the process to continue its execution as if nothing wrong had happened. `notify(message, address)` is used to inform a stakeholder that something has gone wrong[6]. `halt` simply stops the process in execution. `retry(times)` declares that the system should retry to invoke the web service up to a given number of `times`. `store`, the same as in WSCoL, takes a value and stores it to the persistent storage, making the datum available to future monitoring and recovery activities.

The second group comprises recovery actions that alter the amount of supervision being performed either by modifying the supervision parameters or the supervision rules themselves. `changeSupPar(params)` modifies the supervision parameters associated with the supervision rule being considered. If the supervision rule ever needs to be re-considered in the process instance, the new supervision parameters will be taken into account (for example, when the rule is expressed within a loop). `changeProcessPriority(val)` modifies the global priority level. This allows us to dynamically modify the amount of supervision activities to be considered during execution from that point of the process on. Indeed, through this recovery action we can decide to disable or enable entire groups of rules. `changeSupRules(monitoring, recovery)` modifies how supervision (for the operation at hand) is achieved, and therefore relaxes or tightens the constraints. The compulsory `monitoring` parameter replaces the old monitoring expression with a new one. The definition of a new `recovery` strategy is optional.

The third group changes the services with which the process does business. `rebind(wsdl, operation, xslt)` indicates that the service being invoked is to be substituted with another service. The only required parameter is `wsdl`, which indicates where the new service can be found. If the new service presents a different interface, the designer must also pass parameters `operation` and `xslt`. The first indicates the name of the remote method to be called, and the second explains how to resolve the differences between the message types used by the service being substituted and the new one. Currently, this action does not consider automatically the intrinsic discovery problem that must be resolved to provide a substitute URI. `rebindPartnerlink(name, wsdl, xslt)` is similar to the `rebind` action, except its effects are not limited to the operation being called, but are extended to the entire process in execution. Indeed, we change the endpoint associated with a particular BPEL partnerlink (indicated by parameter `name`) with address `wsdl`. If we are changing to a service with a different interface, we use parameter `xslt` to resolve the differences.

The fourth group contains general actions that do not fit into the first three groups. `call(wsdl, operation, ins, xslt)` consists of a call to an external web service. The first two parameters are used to identify the service (through a WSDL URI) and the name of the operation that we want to call. The third parameter (`ins`) represents the data that are to be sent to the service. The external service being called does not share the same data space as the process, and therefore we use a copy-by-value technique to pass it relevant parameters. The final parameter (`xslt`) is optional and is used to map the service's return message onto the message expected by the business process. `callback(eventHandler, input)` is, potentially, the most

[4]Since we deal with finite sets of data these constructs do not actually add expressive power to the language, but have been included for convenience.

[5]Whose values would be collected while executing the recovery directives.

[6]We currently implement email notification, but other methods could easily be added (e.g., SMS notification).

disruptive of the recovery actions, since it allows direct access to the internal state of the process. This action allows complex logic, embedded in the process by means of an event handler (`eventHandler`), to be used as recovery. When we send it an event, the handler executes in an independent thread with respect to the main business logic, which meanwhile continues to remain synchronously blocked, since it is waiting for an answer from the supervision framework. Once the event handler thread completes, the supervision framework is warned to unblock the main business process. The disadvantage is that event handlers must be statically embedded into the process prior to deployment, meaning that the recovery logic is defined once and for all, and that it can only be personalized through the parameterization of the event handler itself. This is somewhat similar to the current approach held by the BPEL specification, which requires that compensation be defined statically at design time. `restore(destLocation)` takes the process back in time, to the point of execution immediately prior to the `destLocation` (indicated with an XPath expression), and resumes the process execution from there. This is a powerful option, but as we shall see, its actual applicability is constrained by many different factors (e.g., the actual process location, the nature of partner services, and their Web methods.)

B. Recovery Strategies

Designers can create multiple recovery strategies by mixing atomic actions. WSReL is reminiscent of rule-based approaches, and allows us to choose the more suitable course of action, depending on what is going on in the process and in the surrounding environment. The syntax for defining strategies is defined as:

```
⟨strategy⟩ ::= try {⟨step⟩} (elsetry{⟨step⟩})* (else{step})?
⟨strategy⟩ ::= if(⟨condition⟩) {⟨strategy⟩}
               (elseif(⟨condition ⟩) {⟨strategy⟩})*
               (else{⟨strategy⟩})?
⟨step⟩     ::= (⟨action⟩)+
```

where condition is a WSCoL expression or a special keyword `NoResp`, and action is an atomic action taken from those presented in the previous section. The special keyword `NoResp` is true if a service invocation did not answer before the timeout was reached.

A strategy is a sequence of steps. Each step is wrapped in a `try` block, and contains an ordered list of atomic actions chosen from those presented previously. The semantics of the sequence is that first we try to fix the anomaly by executing the atomic actions contained within the first step. If at least one of these actions requires monitoring to be re-enacted, we do so to verify if the anomaly persists. If we are not successful we try with the second block, and so on. To facilitate the definition of these steps, each is executed by considering the original state of anomaly, as if no other block execution had been attempted until then.

We also allow designers to define more than one sequence so that the system can choose the most appropriate one at runtime. The syntax allows us to specify alternative branches that are chosen by checking WSCoL expressions. The order of the `if-elseif-else` branches determines the order in which conditions are evaluated, and how the overall recovery will play out.

Amongst the various alternative branches, designers should remember to treat the case in which the system captures an invocation timeout. Note that, in this case, all post-condition monitoring activities are skipped entirely, and the same is true for any associated data collection. This means that the recovery the designer defines cannot assume that certain data be available. If no explicit recovery for the timeout is provided, the default behavior is to terminate the process.

C. Some constraints

Although WSReL is built for maximum flexibility, there are some constraints we must keep in mind when building a recovery strategy. We must consider:

- Whether the recovery is associated with a pre- or a post-condition. Some actions only make sense in post-conditions (e.g., action `retry`).
- Whether the recovery is dealing with stateful or conversational services. In these cases, certain atomic recovery actions (`retry, rebind` and `changePartnerlink`) should not be used light-heartedly. In fact, re-calling a service might not even be an option, while actions `rebind` and `changePartnerlink` could cause problems if used when the process is in the middle of a conversation.
- Which actions require that monitoring be re-enacted to discover if they were successful in fixing the anomaly. Note that some actions `ignore, notify, halt`, and `call` can always be considered successful.

In general, we have decided for as much freedom as possible, but also for the designer being responsible for the consistency of the different recovery activities.

Backward recovery is an issue all in itself. A `restore` does not require monitoring to be re-performed to see if it was successful, but unless the anomaly is transient we need a way to prevent it from re-occurring. The simple re-execution of a past fragment of the process does not guarantee that the anomaly be avoided. We might need to combine the `restore` with some other actions (e.g., a `rebindPartnerlink`). A `restore` used in conjunction with other atomic actions requires that all actions placed before it be executed in the state in which the anomaly occurred, and all actions afterwards be executed in the destination state[7]. The process is only allowed to restore its execution once the entire recovery step has been completed.

Stateful and conversational services constrain the set of activities that can be considered acceptable destinations: re-invoking a stateful service might not be a pursuable option, and partner services might not allow us to restore a conversation to an intermediate point. We must consider what the process defines between the source and the destination locations. If all the interactions of the fragment are with stateless services, we can perform a `restore`. If partner services are stateful, we must know whether they can be freely re-invoked (e.g., it might not be a good idea to re-invoke a bank payment service!). If there are conversational services, we must be sure not to break the conversation. Finally, if there are BPEL activities that are waiting for an asynchronous message (i.e., a *receive*), we must be sure that the `restore` does not cause them to wait forever. In fact, if the asynchronous message has already been received once (in the first execution of the fragment being restored), it is not obvious that the partner service will send it again.

[7]We define the *source* location as the point in which the anomaly was detected, and the *destination* location as the point from which the process will resume execution after the recovery is completed).

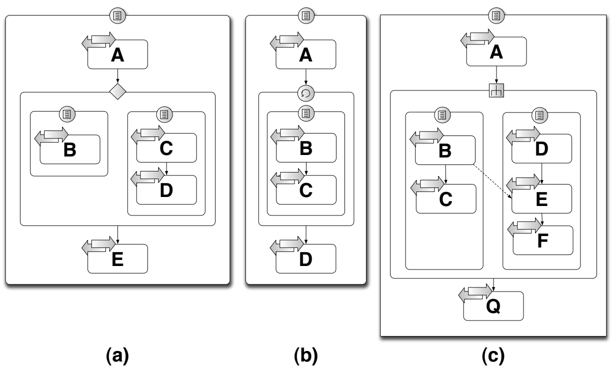


Fig. 3. Scopes for (a) if, (b) while, and (c) flow constructs.

We must also consider what happens when we want to restore a BPEL process with parallel or alternative execution paths (constructs *pick*, *if*, *foreach*, *flow*, *while*, or *repeatUntil*). To do this, we must recall the hierarchical definition of BPEL processes, where every activity in a process has an enclosing scope, be it explicit or implicit. `Restore`s require that a destination be always in the past with respect to its source location (as to execution flow), and be either in the same scope as the source or in a recursively enclosing one. This guarantees that we choose a destination activity executed in the past.

For example, Figure 3(a) shows the implicit scopes associated with a BPEL *if*. If we choose `E` as our source location, the only valid destination is `A`. All other destinations would require us to descend into other scopes. If we choose `D`, `C` is a valid destination since it is in the same scope as the source, but also `A` is acceptable since it is part of a parent scope. On the other hand, `B` is not valid since we would need to exit the scope enclosing `D` and re-descend into a different one. Similarly Figure 3(b) shows the scopes of a *while*. If we choose `D` as our source, the only valid destination is `A`. We cannot choose to restore `C` since we would need to descend two scopes to reach it. On the other hand, if we choose `C`, both `B` and `A` are valid destinations. When dealing with loops, the execution of a `restore` does not allow us to change loop iteration. It always takes us back to before the most recent past execution of the destination activity. Therefore, if we choose `B` as our destination, the process is restored to before the execution of `B` in the same iteration as the execution of the activity starting the reaction.

BPEL supports concurrent paths through its *flow* control structure, and activity synchronization through *link*s. Figure 3(c) shows an example in which there are two concurrent sequences. Synchronization links (shown as dotted lines) indicate that `E` cannot run before `B`. Again, a destination must always be in the past, and either in the same scope as the source or in a recursively enclosing one, but now we also need to consider the activities that need to be re-executed because of synchronization links. The optimal case —no synchronization links— imposes that we only re-execute the activities that are in the sequence that leads from the destination up to the source. We do not need to re-execute the other threads since nothing has changed. For example, if there were no synchronization links in Figure 3(c), and our source and destination were `E` and `A` respectively, we would re-execute activities `A`, `D`, and `E`; the other thread is independent.

If we need to re-execute an activity with an outgoing link, we need to consider the semantics of BPEL links. Indeed, its re-execution (e.g., `B` in Figure 3(c)) causes a re-evaluation of the *transitionCondition* at `B`, and therefore of the *joinCondition* at `E`. This impacts the executability of `E` and following activities. Therefore, our re-execution follows the synchronization links as well.

The case of incoming links is slightly different. A *transitionCondition*'s value is set after the activity it originates from has completed. Apart from what we can learn from synchronization links, there is no way to know the order in which the activities in the various flow branches execute. Therefore, if we are not re-executing the activity from which the link originates, there is no reason to believe that the activities we are re-executing will impact its *transitionCondition*. It continues to contribute to the *joinCondition* as it did in the previous execution. This means that incoming links do not, by themselves, impact the amount of activities we need to re-execute. For example, if we perform a restore, and our source and destination are `F` and `E` respectively, the contribution of `B`'s outgoing link to `E`'s *joinCondition* remains the same, so the incoming link is not followed and no extra activities need to be re-executed.

The final comment concerns *forEach* structures. If source and destination activities are both within the structure body, and we assume the serialized execution of internal activities, it is as if we were in a loop. In the case of parallel executions, since the standard does not consider synchronization links, there is no way to distinguish among the different threads, so we simply restore the statement as if it were atomic.

VI. Examples

Now that supervision rules are fully defined, we can turn the supervision requirements introduced in Section II into proper rules. The first example gives a complete supervision rule, then we avoid expressing the *location* and the *supervision parameters* since they are straightforward.

CheckReservation is rendered as a post-condition for *invoke* activity `storeDate`, and we state that the reservation number be a 7-character code, of which the first character is "N" and the last six are digits. To express this assertion we need to use data type-specific functions. The `length` function gives us the number of characters in the code, while the `substring` function allows us to extract the six characters that need to be digits. Alias `$length` has to be 7, the internal variable has to start with N, and `$sub` has to be between 0 and 999999. If an anomaly arises, we restore the execution to before activity `getCalendarDates` (`getDatesXPath` is the XPath expression that uniquely identifies *invoke* activity `getCalendarDates`), and we change the partner service. The effect will be to try to get a reservation through a backup appointment manager service.

```
Location:
/process/pick/onMessage[1]/invoke[4]

Priority: 3

Expression:
let $res = $StoreReservationResponse/code;
let $length = ($res).length();
let $sub = ($res).substring(1,6);
$length ==7 && ($res).startsWith('N') &&
$sub > 0 && $sub < 999999;

Reaction:
try {
   restore(getDatesXPath);
   rebindPartnerLink('appointmentPL',
   'http://AppointmentManager2',null);
}
```

CheckCenter becomes a pre-condition for *invoke* activity `sendImagesToTMC`. To calculate the service's reliability we need to keep track of its behavior over time. This can be achieved by using an additional post-condition to store certain values for future use. In our case we store the service's response time:

```
Expression:
let $rt = return('RT_URI',
   '<getResponseTime><processID>TeleRadiology
   </processID><instanceID>' + $instanceID +
   '</instanceID><activity>//invoke[@name =
   'sendImagesToTMC']</activity><iteration>0
   </iteration></getResponseTime>',
   /getRTResponse/ms);
store $rt;

Reaction:
let $rt = -1;
if (NoResp) {try {store $rt;}}
```

The service's response time is obtained using a special-purpose external probe. We call the probe stating that we are interested in the response time of the last iteration of activity `sendImagesToTMC` (the same activity can be executed more than once in a process). Notice that we use special alias `$instanceID` to obtain the ID of the process instance being run. Regardless of its value, `$rt` is stored to the persistent storage. We also add special reaction code for coping with cases in which *invoke* `sendImagesToTMC` fails to respond. In this case, there is no response time so we manually set the value of `$rt` to -1 before storing it.

The values stored in the post-condition can then be used in a pre-condition to calculate an up-to-date reliability value. This can be expressed as:

```
Expression:
let $times = retreive('rt', null, 2h);
let $trues = numOf($t in $times, $t>0 &&
             $t<12E4);
let $total = numOf($t in $times, true);
$total == 0 || $trues/$total < 0.95);

Reaction:
let $URI = $telecenterEndpointReference;
let $now = return('Clock_URI','<getCurrentTime>
            </getCurrentTime>',
            /getTimeResponse/ISO8601);
if ($URI=='http://firstCenter/Center/
            CenterBean?wsdl') {
   try {rebind('http://secondCenter/Center/
        CenterBean?wsdl'),null}}
else {
   try {changeSupParams('<newValidity><from>null
        </from><to>' + $now + '</to>
        </newValidity>')}}
```

First we retrieve the `$rt` values stored in the last 2 hours (`$times`). Then we use function `numOf` twice to calculate how many values are lower than 2 minutes (`$trues`), and the total number of retrieved values (`$total`). Finally, the up-to-date reliability is given as `trues` over `$total`.

Since the assertion is a pre-condition, we react by changing the binding with a backup service, before performing the invocation. However, we only do this if the process is still bound to the original telecenter service. If the binding has already changed (for example through event handler `changeBinding`) we switch this supervision rule off by changing its validity parameter. This is done by setting the validity's `to` value to the moment in time in which the recovery takes place (`$now`).

The difference between performing the `rebind` explicitly and using the process' event handler (as in *CheckTimelyAnalyses*) is that the former only has a temporary effect while the latter is permanent. In fact, in the former we change the binding solely for the monitored invocation. This is transparent to the rest of the process. On the other hand, if we change the binding through the event handler, it remains modified until we change it again.

CheckResolution is seen as a post-condition for *Receive* activity `receiveMRI`, where we use an external service to calculate the image's horizontal and vertical resolutions. We use two variable aliases to simplify the overall condition, which states that the image's resolution must be between 800×600 and 1024×768 pixels. If there is an anomaly, we have different strategies amongst which to choose. If the resolution is too high we can filter it down by calling an external component that offers image modification services. The result is then mapped back onto the message received from `receiveMRI` using an XSLT, and is transparent to the business process. If the resolution is too low, but not too bad, we simply ignore the problem and let the process continue. Finally, if it is truly too low, we notify the head doctor and halt the execution. This can be expressed as:

```
Expression:
let $imageURL = $submitImageRequest/imageURL;
let $hRes = return('WSDL_URI','<getInfo>
            <imageRes>'+$imageURL+'</imageRes>
            </getInfo>',
            /getResResponse/horizontalRes);
let $vRes = return('WSDL_URI','<getInfo>
            <imageRes>'+$imageURL+'</imageRes>
            </getInfo>',
            /getResResponse/verticalRes);
$hRes >= 800 && $hRes <= 1024 &&
$vRes >= 600 && $vRes <= 768;

Reaction:
let $high = $hres >1024 || $vRes > 768;
let $low = $hRes >= 800 * 0.9 &&
           $vRes >= 600 * 0.9;

if ($high == true) {
   try {call('http://imageModifier:8080/
        imageTools?wsdl',
        '<changeRes><image>'+$imageURL+'</image>
        <desiredResolution><hRes>1024</hres>
        <vRes>768</vRes></desiredResolution>
        </changeRes>', XSLT)}
}
elseif ($low == true) {
   try {
     notify('Resolution too low - low error',
            headPhysician@radiology.com');
     ignore();}
}
else {
   try {
     notify('Resolution too low - high error',
            headPhysician@radiology.com');
     halt();}
   }
}
```

VII. Supervision Framework

Supervision rules are supported by a dedicated framework built on top of ActiveBPEL [16], a well-known open-source BPEL

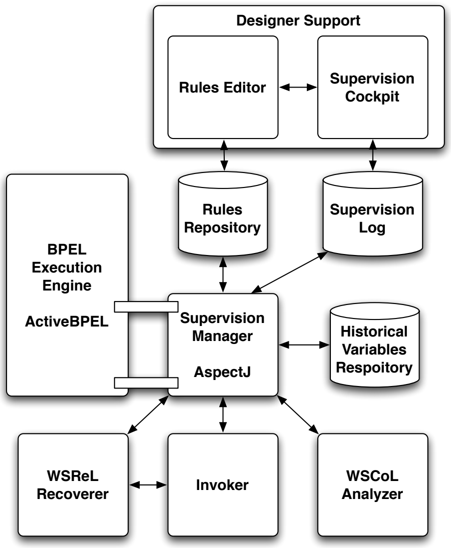


Fig. 4. The Supervision Framework.

execution engine, to keep the compatibility with standard technologies and to complement them with supervision capabilities.

Typically, supervision requirements depend on the needs of different stakeholders, on the process' life-cycle, and on the environment in which the process is run. This means that we cannot statically associate a single set of supervision rules with a process, and that the supporting framework needs to continuously obtain run-time information to choose the exact set of rules to consider as the process executes. The context evolves, supervision parameters can dynamically modify the amount of supervision being performed, and also the rules themselves can change at runtime.

These requirements led us to choose aspect oriented programming (AOP [11]) as the principal enabling technology[8]. We use AOP to treat supervision as a true cross-cutting concern and to centralize its management. Differently from other approaches [17], we decided to apply AOP to the executor, and not to the BPEL process itself, to only deploy the business logic once and be able to blend supervision capabilities as late as possible. Appropriate join points in the ActiveBPEL engine let us intercept the process' execution, and perform the entire supervision loop. Roughly, the advice code gathers run-time information, chooses the appropriate rules to consider, evaluates their supervision parameters, and if required checks them and reacts as needed.

Figure 4 shows the architecture of the our supervision framework. It consists of two sets of components: those dedicated to designers, to define supervision rules and visualize the results of their evaluation, and those dedicated to evaluating monitoring statements and executing recovery actions. The former consist of the *Rules Editor* and the *Supervision Cockpit*. The *Rules Editor* is a web-based tool that reads the definition of a BPEL process, designed using ActiveBPEL's own designer tool, and helps the user assemble supervision rules by providing all the locations where rules can be added, all the BPEL variables that can be converted into WSCoL variables, all the external probes already deployed and available for external variables, and all the WSCoL and WSReL constructs that can be used in the different parts of a rule. The result is stored in the *Rules Repository*. The *Supervision Cockpit*, on the other hand, provides different means to visualize the *Supervision Log*, which contains the history of all the supervision activities that have taken place, to let the designer get a summary of how the different process instances execute. It can also be used to modify the actual priority with which the different instances are supervised, and to change the rules themselves on-the-fly.

In the latter set, the *ActiveBPEL execution engine* is the key component. Since it is written in Java, we used AspectJ [18] to add supervision capabilities through aspects. The *Supervision Manager*, which is our true AspectJ advice, is responsible for managing the enactment of supervision rules, which are extracted from the *Rules Repository*. In particular, it gathers the values of internal variables from the engine, those of external variables from deployed probes, and those of historical variables from the *Historical Variables Repository*. External variables are obtained through the *Invoker*, which is a general-purpose dynamic Web service invocation tool.

An interesting example of external probe is the one we developed to gather service execution times. The probe is embedded in ActiveBPEL, using the same AOP mechanisms described for activating supervision, and exploits the process engine's clock to store a timestamp before and after any invocation. We can also use probes to transform data we already have, or to obtain further data that depend on what we already know. For example, in Section VI, we used a probe to get image resolutions. This is achieved by implementing a special purpose service, a practice that can be repeated every time no appropriate probes are already available.

The actual monitoring and recovery activities are then delegated to the *WSCoL Analyzer* and to the *WSReL Recoverer*, respectively. All the activities are logged onto the *Supervision Log* both for feeding the *Supervision Cockpit* and for off-line analysis.

To understand how the *Supervision Manager* works, we need to introduce ActiveBPEL's internals. The engine creates a *Definition Tree* for each deployed process (regardless of the number of instances running concurrently), and an *Implementation Tree*[9], which extends the previous one, for each new instance. To execute a process instance, the engine traverses the *Implementation Tree* and calls the `execute` methods defined for each node type. For example, an *invoke* node uses the engine's AXIS infrastructure to call a partner service. This is to explain that the *Supervision Manager* intercepts all the calls to the `execute` methods on nodes representing *invoke*, *receive*, *reply*, and *pick* activities, that is, all the BPEL activities that interact with external services. Note that we also intercept the process every time it launches a service invocation timeout exception. This way we can stop the

[8]AOP is a programming paradigm that helps increase a program's modularity by improving separation of concerns, especially when they "cut across" different abstractions in the code (e.g., modules, classes, or methods). Logging is an example one of these concerns: its management requires dedicated code usually tangled with the actual business logic. AOP provides means to encapsulate this code, centralize its management, and mix it with the business logic seamlessly. Typical AOP jargon refers to *advice*, *join points*, and *pointcuts*. An advice is a portion of code that implements the behavior we want to add to our application. Join points represent the locations in our application at which the advice code is activated. Pointcuts are used to define the patterns that detect whether a join point has been reached or not.

[9]These two terms, *Definition* and *Implementation* trees, do not exist in the standard ActiveBPEL terminology, but are introduced here for a clearer presentation.

exception from propagating under the form of a BPEL fault, set the `NoResp` flag, and properly activate recovery.

A. Performance evaluation

Our approach is time consuming by nature, and a performance hit is to be expected. In fact, every time we reach an *invoke, receive, reply,* or a *pick*, we stop the process momentarily and check whether there is a supervision rule that needs to be considered. This operation is time consuming, even if we end with no rule to check. Although supervision consists of both monitoring and recovery, the former undoubtedly represents the key factor when we consider performance. Monitoring is always performed, while recovery is only activated when the process presents an anomaly, and the overhead can be seen as the price to pay to fix the problem.

To evaluate the actual impact the approach has on the execution of a standard BPEL process, we set an in-lab experiment. The test process mimics the eHealth example of Section II-B and uses special-purpose services developed by us. External (real) services were not needed (and too complex) to address the worst case scenario, that is, the situation in which the supervision infrastructure has the highest impact on the overall execution: a controlled environment, in which there were no network problems or delays since all services were run on-site. We did not want to measure the response time of the partner services, but the impact our framework has on the process' execution.

The framework was run on a 1.83 GHz Intel Core Duo with 2 GB of RAM. We considered some 1000 process executions, for a grand total of 6984 BPEL activities. We analyzed the time our system takes to perform the supervision rules presented in the previous sections. First we evaluated the time the Supervision Manager takes to stop the process and check for a supervision rule. This amounts for the time lost, regardless of the presence of supervision rules. The average time spent on this activity was $30ms$. We tried executing the process without any supervision rules, and calculated an average execution time of $300ms$ for each activity. Therefore, if all services are run locally, our modification can be accounted for $1/10$ of the time. If services were to be run on a network, and were real services, the actual impact would be proportionately less since we would need to take into account the delays introduced by the network when the process interacts with its partner services. Besides this, there are two main aspects that drive monitoring execution time: the nature of the collected data, and the complexity of the expressions that need to be checked.

Table I summarizes the data collection times (given in milliseconds) of our experiments. Internal variables are by far the ones that cost less. The reason is that they are the only kind of data that can be collected without invoking external components. On average, the extraction of a historical variable will cost six times as much, while external variables represent a completely different issue. External variable are subject to network issues, and to the actual amount of work the probe has to achieve before returning the results we need. In our experiments the remote services were kept simple and placed within a controlled environment, keeping their response times low. However, caution must be used when choosing external probes, to avoid high performance hits when performing supervision.

TABLE I
COLLECTION TIMES FOR WSCOL VARIABLES.

Variable type	NUM	Collection time		
		avg	min	max
Internal	576	18	13	33
External	480	80	52	491
Historical	40	110	82	612

Table II gives a breakdown of the execution times for each rule. For each we have extracted the number of executions (NUM), the monitoring time, the recovery time, the global time, which includes both supervision and the execution of the actual BPEL activity, and the average slowdown due to supervision. The number of executions for each supervision rule varies due to the fact that they were run using different supervision parameters, and that as the experiments proceeded some activities were dynamically switched off by recovery. *CheckReservation* is the fastest rule to monitor, since it only considers a single internal variable, and a property that uses simple functions. *CheckResolution* is much higher since it introduces the use of two external variables. Finally, we have *CheckCenter*'s pre-condition, which uses a historical variable, as well as two aggregate functions.

As for recovery, *CheckCenter* and *CheckResolution* both present more than one possible strategy. To present more consistent and interesting performance times, we have decided to only consider *CheckCenter*'s second strategy, and *CheckResolution*'s first strategy .

In our experiments we noticed that actions `retry`, `call`, and `callback` can be taxing since they all require an external service to be invoked. In this sense they contribute to the recovery time much like external data collection does to monitoring time, meaning the same caution should be adopted when using them. Activities `changeSupParams`, `changeProcessPriority` `changeSupRules`), and `store` interact directly with the Rules Repository or the Historical Variables Repository, and the order of magnitude of their contribution is similar to that of historical data collection. Actions `rebind`, `rebindPartnerlink`, and `restore` are faster since they are resolved locally to the process execution. Indeed, their contribution is in the order of magnitude of internal data collection. Finally, activities `notify`, `ignore`, and `halt` cause negligible overhead (i.e., less than 15 milliseconds).

Amongst our examples, *CheckReservation* is the fastest since it is resolved locally. It is followed by *CheckResolution*, which requires further service invocations while performing recovery. Finally, we have *CheckCenter*. It's recovery time is driven up by the extra data collection it performs (one internal and one external variable) before changing the rule's supervision parameters.

Note that although these data were collected on a particular process and in a very constrained environment, they can be considered good representatives to assess the impact the different parts of the framework, and the languages' constructs, have on running processes. These figures do not consider the usual delay introduced by the network, neither do they take into account the execution time of partner services. They provide a significant set of unbiased measures of the delays.

B. Lessons Learned

Besides our in-lab experiments, the supervision framework has been used and evaluated by industrial and academic part-

TABLE II
EXPERIMENTAL RESULTS.

Rule	NUM	Monitoring			Recovery			Global			Slowdown
		avg	min	max	avg	min	max	avg	min	max	
CheckReservation	120	32	27	43	39	25	58	332	320	350	1.27
CheckCenter	26	164	149	177	110	104	122	528	508	546	2.07
CheckResolution	240	134	94	722	47	40	94	438	386	1014	1.77

All values are given in milliseconds, except for column slowdown. It shows the ratio between the time required to execute the supervised BPEL activity and the time taken to execute the same BPEL activity without supervision.

ners, both in EU-funded and national projects, (e.g., SeCSE[10] and Discorso[11]). For example, in the SeCSE project [19] the framework was evaluated by more than twenty users throughout a three-step process. At the end of each step, the users were required to fill out a questionnaire. During the first evaluation, the users were assisted in the use of the framework, given the relative immaturity of the prototype implementations. The second and third evaluations were conducted using progressively refined versions of the framework, and were performed by the users without direct assistance. Industrial partners tried the framework on their demonstrators, while colleagues and students applied it onto example processes like ours.

During these evaluations the following benefits and limitations emerged. The first benefit is that we provide a *complete and coherent supervision framework*. Monitoring and recovery are not two separate issues, but can be planned together. The designer's job is made easier by the fact that the same abstraction level is maintained across WSCoL and WSReL. A second important benefit is that the framework adopts a *holistic approach to data collection*. WSCoL's distinction between internal, external, and historical variables gives us great flexibility when designing supervision. In particular, external and historical variables allow us to consider context in a very broad sense. The designer can indeed extend the analysis to whatever notion of context, provided that the necessary probes have been deployed. A third important benefit consists in the *broadness of our languages*. On the one hand, WSCoL provides all the basic elements needed to express a wide variety of properties (e.g., temporal and stochastic properties, or key performance indicators). On the other hand, WSReL provides an easily extensible set of small and simple atomic recovery actions. In both cases we have chosen to support small building blocks. The result is that the languages provide a high degree of programmability since blocks are easy to aggregate. However, particularly in WSReL, there is the risk that the number of blocks can become too high, making it hard for the designer to decide which blocks to use in a particular supervision. A fourth benefit is that the framework supports *standardization and separation of concerns*. In our approach supervision is not integral to the overall design of a process' business logic. The actual BPEL code and the supervision rules are kept separate at design time, and only inter-weaved at runtime. Although supervision requirements may evolve over time, what is actually deployed to the execution framework is a 100% BPEL compliant process, and there is no need to remove, modify, and redeploy it just to change how supervision is performed. Finally, the framework can easily be used, as we have already experimented, as a *backend* for frameworks for probing business-level key performance indicators, for SLA and policy management [20], or even for high-level business rules [21].

None of our users complained about the performance degradation introduced by supervision. Even if in our experiments the impact was not negligible, the actual execution times of the partner services usually hide the delay and the end-to-end (user-process) response times are only slightly touched.

Regarding the limitations of our approach, and in particular of our languages, we found that supervision parameters were used only partially. Although it is possible to combine different supervision parameters to act as a "virtual knob" for selecting the amount of supervision activities to perform, our partners concentrated on parameter *Priority*, since it is the easiest to understand. Many of the users said that they do not need to switch rules on and off according to time constraints and that since they do not use dynamic binding strategies, the idea of having trusted providers is good and interesting, but currently useless for them.

Our subjects also preferred WSCoL to WSReL. Since the two languages are similar, we interpret this as them being more interested in monitoring than in sophisticated recovery solutions. Some of our users found it difficult to combine atomic recovery actions to obtain the results they wanted, especially when dealing with the additional complexity introduced by backward recovery.

All our users tended to concentrated on subsets of the proposed notations. Different needs call for different aspects of the languages, but the feeling (lesson) is that average users are interested in particular problems and as soon as they solve them they are not interested in many of the details. This finding can also be read as the need for more detailed guidelines, and high-level abstractions to guide users while defining their supervision rules. Currently, our languages call for users who are good at BPEL and XML technologies, but suitable supervision *patterns* a-la [22], or a simple graphical language for combining building blocks to create complex rules, while enforcing their consistency, would widen the target audience and improve the proposal's usability. Some of the users asked for predefined solutions for common QoS dimensions, others for statistical macros, that is, WSCoL expressions/extensions for modeling statistical functions. This would also allow us to improve the *Rules Editor*, to help designers reach their goals more easily.

VIII. RELATED WORK

Many of the problems we tackle in this article have also been confronted in other research communities. Possible examples are

[10]SeCSE (Service Centric Service Engineering) is a EU integrated project on the specification, discovery, design, and management of services. More information is available at: `www.secse-project.eu`.

[11]Discorso (Distributed Information Systems for COoRdinated Service Oriented interoperability) is in Italian project on the use of service-based infrastructures to foster and support the creation of virtual districts and enterprises. More information is available at: `www.discorso.eng.it`.

TABLE III
COMPARISON OF MONITORING APPROACHES.

Approach	Language		Abstraction		Properties		Directives			Timeliness	
	Logic	HL/VHL	Domain	Implementation	Safety	Temporal	Process	Activity	Event	Synchronous	Asynchronous
Sahai et al.		x	x			x	x				x
Keller and Ludwig		x	x			x	x				x
Skene et al.		x	x			x	x				x
Erradi et al.	?	x		x	x	x		x		x	x
Pistore et al.	x			x	x	x	x		x		x
Mahbub and Spanoudakis	x			x	x	x			x		x
Moser et al.		x	x		x			x			x
WSCoL	x			x	x			x		x	

Language indicates the type of specification used by the approach (logic or HL/VHL), abstraction indicates the abstraction level at which properties are defined (domain or implementation), properties is used to indicate the kind of properties definable by the language (safety or temporal), directives indicates the level at which a property can be evaluated (process, statement, or activity), timeliness indicates when the monitoring activity is performed (synchronous or asynchronous).

transactions in database systems [14], [15] and workflows [4]. Further examples can be found in the communities that study the context-aware engineering of web applications [23], the run-time verification of component based systems [3], [24], [25], the modeling and deployment of dynamic and self-adaptive software architectures [26], [27], and the coordination of critical grid-based systems [28]. Given the scope of the article, in the following we concentrate on service oriented solutions, and give an insight to the ones we believe to be more promising or interesting. There are not many approaches that offer integrated solutions to both monitoring and recovery, and this already sets our proposal apart from most of the work in the literature. For ease of presentation we first tackle work that mainly concentrates on run-time monitoring; then, we consider recovery approaches, or more in general solutions for the run-time steering of service-based applications.

A. Monitoring

Table III gives a detailed comparison among our main competitors as for monitoring. The classification of the approaches follows the taxonomy presented by Delgado et al. in [29], with some modifications/extensions of the metrics to adapt them to the service-oriented context.

In the field of service monitoring, many of the first works concentrated on the notion of Service Level Agreement (SLA). Sahai et al. [30] described an automated and distributed SLA monitoring engine. Keller and Ludwig [31] proposed a framework to define and monitor SLAs that focus on QoS properties such as performance and costs. Skene et al. [32] proposed the SLAng language for SLAs, described using meta-modeling techniques. The three approaches differ greatly in scope from our own. They establish high-level abstract specifications and concentrate on QoS, so that specific agreements can be negotiated between the service provider and the consumer. Our approach works at a lower-level of abstraction, nearer to the actual implementation and to the designer's needs. As stated in Section VII-B, our approach could easily be adapted to provide the backend for a SLA-based framework. A second big difference lies in the nature of the data considered during the monitoring activities, since we go beyond simple QoS and stress the need for a holistic approach that considers functionality, QoS, and environmental data. Context and situational aware applications are becoming more present in real-day life, and we believe that our decision to consider context information is an important one.

A second line of research revolves around WS-Policy [33], which allows providers to specify their provisioning policies, and clients to specify their requirements. The standard itself does not suggest any general-purpose language for defining policies. However, Erradi et al. [34] have proposed an extension called WS-Policy4MASC, allowing designers to define the source of the monitoring data (they support both internal and context data), the modality of the monitoring (synchronous or asynchronous), meta-level information similar to our supervision parameters, and the actual properties. The authors provide a .NET based implementation that requires the processes be run using the Microsoft Workflow Foundation. Their work is in many ways similar to ours, but they do not clearly define, nor give examples, of the language they use for specifying the properties, making it hard to understand its expressive power and its actual details, and to provide a deep comparison with our work.

Mahbub and Spanoudakis [35] propose a framework for the validation of behavioral properties. These properties are expressed using event calculus. Their monitoring is performed asynchronously, and event calculus is used to define their monitoring properties. A first comparison with our approach can be made by looking at the actual languages proposed. Event calculus, with its explicit temporal operators, make it easier to specify certain QoS-oriented monitoring activities. Although these properties can be expressed in WSCoL, they require special-purpose external probes that can introduce further performance loss. This ties into our desire, as stated in Section VII-B, to study high-level abstractions, or monitoring patterns, for dealing with the monitoring of common QoS properties. Another important factor in the comparison between the two approaches is that we provide a synchronous approach while they provide an asynchronous one. On the one hand this gives them an advantage in terms

TABLE IV
COMPARISON OF RECOVERY APPROACHES.

Approach	Language			Location		Actions					Data Source	
	Programming	Logic	HL/VHL	Instance	Proxy	Retry	Substitute	Compensate	Restore	Others	Process	External
Ardagna et al.			x		x	x	x	x			x	
Colombo et al.	x		x		x		x				x	
Moser et al.			x		x		x				x	
Charfi et al.			x	x				x	x		x	
WSReL		x		x		x	x	x	x	x	x	x

Language indicates the abstraction level at which the designer must define the recovery (Programming, Logic, or HL/VHL), Location indicates "where" the recovery is achieved (within the process Instance or within a Proxy), Actions defines the types of recovery supported by the approach (Retry, Substitute, Compensate, Restore and Others), while Data Source defines the nature of data used to guide the recovery (taken from the Process or External Data).

of performance overhead since they are less invasive. However, the decision to perform monitoring synchronously allows us to be much more timely in discovering errors, and to provide an integrated framework that also considers recovery. Indeed, their asynchronous approach would need re-synchronization mechanisms to activate corrective measures. But re-synchronization with the process is a difficult issue to solve, and is not even feasible in certain cases. Backward recovery could become a means to fix the errors "where" they occur, but there would always be a mismatch between the current state and the one in which the problem manifested. Indeed, there are currently no integrated supervision approaches that make use of asynchronous monitoring.

Moser et al. [36] present VieDAME, a non-intrusive approach to monitoring. Interestingly, their architecture is based on AOP techniques, and is pluggable with respect to different engines. The approach accumulates data as the process instances are run, aggregating previous data to calculate QoS values such as response time, accuracy, or availability, while intrusiveness is minimized. Comparing it to our approach, the performance overhead is obviously less. However, an important aspect to consider is the limited flexibility they provide in defining what should be monitored. First of all they concentrate on QoS, and do not consider a holistic approach as we do. There is no notion of context, which is important given the current trend in situational and context-aware applications. Our decision was to empower the designer, and to allow him/her to define more complex properties. Performance should then be managed by exploiting our notion of supervision parameters, which allow to dynamically set how invasive the supervision activities truly are.

B. Recovery

Less work has been achieved in the context of complex process recovery. Most of the process recovery approaches, or steering solutions, present in the literature limit themselves to the more simple notion of dynamic binding. These approaches try to update the set of services with which they do business, and provide optimized experiences. As we shall see, some limit themselves to substituting services that offer the same interface, while others provide mediation mechanisms. Once again, as stated in Section VII-B, a main difference with respect to our approach is that we consider flexibility a key aspect. This is why we present an extensible range of very different recovery actions. Table IV gives a detailed comparison of the competing recovery approaches.

Ardagna et al. [37], [38] propose the PAWS (Processes and Adaptive Web Services) framework. Their proxy-based framework optimizes a BPEL process' QoS by selecting the most appropriate partner services at runtime, and by providing a set of simple recovery actions. First, designers define global and local QoS constraints. Second, these requirements are analyzed and used to produce a set of candidate services, retrieved from an extended UDDI repository. Third, the system provides a series of mediations that allows it to deal with retrieved services. If a QoS requirement cannot be met, the framework can choose among a set of recovery actions: retry, substitute, and compensate. Comparing it to our approach we can state that they have a similar attention to separation of concerns. Indeed there is no notion of monitoring, discovery, or mediation to be found in their processes, but these issues are treated externally in proxies that are placed between the process and the partner services. Unfortunately the extensive use of proxies brings a high performance overhead. Moreover, their recovery strategies are defined statically at design-time. The way they implement separation of concerns does not allow them to add, or modify, recovery strategies at runtime, nor does it allow them to select strategies at runtime depending on the actual context of execution.

Colombo et al. [39] offer a composition language that allows designers to declare policy (re)binding rules. Policies are defined using an extended version of the Drools language [40] (a language for defining Event-Condition-Action rules), and can be either global or local. The approach is proxy-based. Every time the process invokes a service, the proxy interacts with the rule engine to see whether (re)binding is necessary. The authors also added mediation capabilities through a special-purpose mediation scripting language and an interpreter that behaves as a proxy [41]. The only recovery action provided is dynamic re-binding, while we empower the designer with a wider set of options. Once again there is a clear separation of concerns between business logic and supervision that is enforced by the use of proxies. However, the definition of recovery strategies is given statically. Moreover, the approach requires that the designer be comfortable with the Drools language, which has a completely different level of abstraction with respect to the business logic.

Moser et al. [36], in their VIeDAME approach, also provide a dynamic adaptation and message mediation service for partner links. Using the data collected during the monitoring step, the system chooses the most appropriate service, while XSLT or regular expressions are used to transform messages accordingly. VIeDAME excels in maximizing simplicity, at the cost of providing a very limited set of possible recoveries. In practice it is limited to dynamic re-binding with negotiation.

Something similar to our backward recovery has been attempted in the context of transactional BPEL processes. Charfi et al. [17] propose an AOP-based solution for enforcing the interaction with the Apache Kandula WS-TX middleware. If a transaction cannot be closed, special purpose fault handlers are used to rollback the process. This means that any notion of recovery strategy needs to be statically coded by the designer in terms of BPEL code. The approach favors separation of concerns for enforcement, but not for recovery. A final note is that the implementation is based on the AO4BPEL engine, and requires that the partner services be WS-TX aware.

IX. Conclusions and Future Work

Service compositions provide unprecedented levels of dynamism and flexibility. Using services exposed by third parties, we construct systems whose ownerships are intrinsically distributed, making it hard to reason about the actual functionality and quality of service we can ensure at runtime. The challenge lies in providing composite systems that are robust and dependable. To this end we blur the lines between design-time and runtime validation, and provide self-supervision to identify and autonomously react to anomalous situations that may occur during execution.

The article has presented one of the few integrated frameworks for both the monitoring and recovery of BPEL processes. WSCoL and WSReL, along with the prototype framework, provide a complete and coherent solution able to address many diverse business domains and user needs. The actual support offered to design supervision rules must be improved and some advanced capabilities (like patterns and predefined templates) would allow us to attract also those users that have a limited technical background, but deep knowledge of the business domain. These issues are part of our future work, in which we will address the feedback we received during the evaluation.

References

[1] Oasis Consortium. [Online]. Available: http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=soa-rm

[2] B. Meyer, "Applying Design by Contract," *Computer*, vol. 25, no. 10, pp. 40–51, Oct. 1992.

[3] F. Chen, M. d'Amorim, and G. Rosu, "Checking and Correcting Behaviors of Java Programs at Runtime with Java-MOP," *Electr. Notes Theor. Comput. Sci.*, vol. 144, no. 4, pp. 3–20, 2006.

[4] G. Alonso and C. Mohan, "WFMS: The Next Generation of Distributed Processing Tools," in *Advanced Transaction Models and Architectures*, 1997.

[5] L. Baresi, E. D. Nitto, and C. Ghezzi, "Towards Open-World Software: Issue and Challenges," in *SEW*. IEEE Computer Society, 2006, pp. 249–252.

[6] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, and D. Ferguson, *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging and More*. Prentice Hall PTR Upper Saddle River, NJ, USA, 2005.

[7] Jordan, Evdemon, Alves, Arkin, Askary, Barreto, Bloch, Curbera, Ford, Goland, Guizar, Kartha, Liu, Khalaf, Konig, Marin, Mehta, Thatte, van der Rijn, and Y. Yiu, "Web Services Business Process Execution Language Version 2.0," 2007, bPEL4WS specification.

[8] D. C. Luckham, F. W. von Henke, B. Krieg-Brueckner, and O. Owe, *ANNA: a Language for Annotating Ada Programs*. New York, NY, USA: Springer-Verlag New York, Inc., 1987.

[9] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll, "An Overview of JML Tools and Applications." *International Journal on Software Tools for Technology Transfer*, vol. 7, no. 3, pp. 212–232, 2005.

[10] B. Meyer, "Applying Design by Contract." *Computer*, vol. 25, no. 10, pp. 40–51, 1992.

[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier, and J. Irwin, "Aspect-Oriented Programming." in *ECOOP'97 - Object-Oriented Programming, 11th European Conference, Proceedings*, ser. Lecture Notes in Computer Science, vol. 1241. Springer, 1997, pp. 220–242.

[12] L. Baresi and S. Guinea, "Towards Dynamic Monitoring of WS-BPEL Processes." in *ICSOC 2005: 3rd International Conference on Service Oriented Computing*, ser. Lecture Notes in Computer Science, vol. 3826. Springer, 2005, pp. 269–282.

[13] ——, "A Dynamic and Reactive Approach to the Supervision of BPEL Processes," in *ISEC 2008: 1st Indian Software Engineering Conference*, 2008, pp. 39–48.

[14] C. Zaniolo, S. Ceri, C. Faloutsos, R. Snodgrass, V. S. Subrahmanian, and R. Zicari, *Advanced Database Systems*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997.

[15] J. Widom and S. Ceri, *Active Database Systems: Triggers and Rules For Advanced Database Processing*. Morgan Kaufmann, 1996.

[16] ActiveBPEL Engine Architecture. [Online]. Available: http://www.activebpel.org/docs/architecture.html

[17] A. Charfi and M. Mezini, "AO4BPEL: An Aspect-oriented Extension to BPEL," in *World Wide Web*, 2007, pp. 309–344.

[18] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An Overview of AspectJ," in *ECOOP 2001 - Object-Oriented Programming, 15th European Conference, Proceedings*, ser. Lecture Notes in Computer Science, vol. 2072. Springer, 2001, pp. 327–353.

[19] SeCSE Consortium. A4.D18 Third Evaluation Report. EU IP Project Deliverable. [Online]. Available: http://www.secse-project.eu/wp-content/uploads/a4d18-third-evaluation-report.pdf

[20] L. Baresi, S. Guinea, and P. Plebani, "Policies and Aspects for the Supervision of BPEL Processes," in *CAiSE*, ser. Lecture Notes in Computer Science, J. Krogstie, A. L. Opdahl, and G. Sindre, Eds., vol. 4495. Springer, 2007, pp. 340–354.

[21] L. Baresi, S. Guinea, and M. Plebani, "Business Process Monitoring for Dependability," in *WADS*, ser. Lecture Notes in Computer Science, R. de Lemos, C. Gacek, and A. B. Romanovsky, Eds., vol. 4615. Springer, 2006, pp. 337–361.

[22] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in Property Specifications for Finite-State Verification," in *ICSE*, 1999, pp. 411–420.

[23] F. Daniel, "A Portable Approach to Exception Handling in Workflow Management Systems," Politecnico di Milano - Dipartimento di Elettronica e Informazione, Tech. Rep., 2006.

[24] M. Kim, M. Viswanathan, S. Kannan, I. Lee, and O. Sokolsky, "Java-MaC: A Run-Time Assurance Approach for Java Programs," *Formal Methods in System Design*, vol. 24, no. 2, pp. 129–155, 2004.

[25] K. Havelund and G. Rosu, "An Overview of the Runtime Verification Tool Java PathExplorer," *Formal Methods in System Design*, vol. 24, no. 2, pp. 189–215, 2004.

[26] J. Dowling and V. Cahill, "The K-Component Architecture Meta-model for Self-Adaptive Software," in *Reflection*, ser. Lecture Notes in Computer Science, A. Yonezawa and S. Matsuoka, Eds., vol. 2192. Springer, 2001, pp. 81–88.

[27] I. Gorton, Y. Liu, and N. Trivedi, "An Extensible, Lightweight Architecture for Adaptive J2EE Applications," in *SEM*, E. Wohlstadter, Ed. ACM, 2006, pp. 47–54.

[28] B. Törnqvist and R. Gustavsson, "On Adaptive Aspect-oriented Coordination for Critical Infrastructures," *Issues on Coordination and Adaptation Techniques*, p. 63.

[29] N. Delgado, A. Q. Gates, and S. Roach, "A Taxonomy and Catalog of Runtime Software-Fault Monitoring Tools," *IEEE Transactions on Software Engineering*, vol. 30, no. 12, pp. 859–872, 2004.

[30] A. Sahai, V. Machiraju, M. Sayal, L. J. Jin, and F. Casati, "Automated SLA Monitoring for Web Services," in *Proceedings of the 13th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management - Management Technologies for E-Commerce and E-Business Applications*, ser. Lecture Notes in Computer Science, vol. 2506. Springer, 2002, pp. 28–41.

[31] A. Keller and H. Ludwig, "Defining and Monitoring Service-level Agreements for Dynamic e-business," in *Proceedings of the 16th Conference on Systems Administration*. Berkeley, CA, USA: USENIX Association, 2002, pp. 189–204.

[32] J. Skene, D. D. Lamanna, and W. Emmerich, "Precise Service Level Agreements," in *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 179–188.

[33] S. Bajaj, D. Box, D. Chappell, F. Curbera, G. Daniels, P. Hallam-Baker, M. Hondo, C. Kaler, D. Langworthy, A. Malhotra *et al.*, "Web Services Policy Framework (WS-Policy)," *Version*, vol. 1, no. 2, pp. 2006–03, 2006.

[34] A. Erradi, P. Maheshwari, and V. Tosic, "WS-Policy based Monitoring of Composite Web Services," in *ECOWS*. IEEE Computer Society, 2007, pp. 99–108.

[35] K. Mahbub and G. Spanoudakis, "A Framework for Requirements Monitoring of Service based Systems," in *ICSOC '04: Proceedings of the 2nd International Conference on Service Oriented Computing*. New York, NY, USA: ACM Press, 2004, pp. 84–93.

[36] O. Moser, F. Rosenberg, and S. Dustdar, "Non-intrusive Monitoring and Service Adaptation for WS-BPEL," in *WWW*, J. Huai, R. Chen, H.-W. Hon, Y. Liu, W.-Y. Ma, A. Tomkins, and X. Zhang, Eds. ACM, 2008, pp. 815–824.

[37] D. Ardagna, M. Comuzzi, E. Mussi, B. Pernici, and P. Plebani, "PAWS: A Framework for Executing Adaptive Web-Service Processes," *IEEE Softw.*, vol. 24, no. 6, pp. 39–46, 2007.

[38] D. Ardagna and B. Pernici, "Adaptive Service Composition in Flexible Processes," *IEEE Trans. Software Eng.*, vol. 33, no. 6, pp. 369–384, 2007.

[39] M. Colombo, E. D. Nitto, and M. Mauri, "SCENE: A Service Composition Execution Environment Supporting Dynamic Changes Disciplined Through Rules," in *ICSOC*, ser. Lecture Notes in Computer Science, A. Dan and W. Lamersdorf, Eds., vol. 4294. Springer, 2006, pp. 191–202.

[40] M. Proctor, M. Neale, P. Lin, and M. Frandsen, "Drools Documentation," *Available on: http://labs. jboss. com/file-access/default/members/jbossrules/freezone/docs/3.0*, vol. 1, 2006.

[41] L. Cavallaro and E. D. Nitto, "An Approach to Adapt Service Requests to Actual Service Interfaces." in *SEAMS '08: Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems*. New York, NY, USA: ACM, 2008, pp. 129–136.

Luciano Baresi Luciano Baresi is an associate professor in the Dipartimento di Elettronica e Informazione at Politecnico di Milano. His research interests are in software engineering, particularly in dynamic software systems, service-oriented applications, and software architectures. Baresi has a PhD in computer science from Politecnico di Milano.

Sam Guinea Sam Guinea is a researcher in the Dipartimento di Elettronica e Informazione at Politecnico di Milano. His research interests include software and service engineering of dynamic systems. Guinea has a PhD in computer science from Politecnico di Milano.