Dynamo + Astro: An Integrated Approach for BPEL Monitoring

Luciano Baresi Sam Guinea
Deepse Group
Dipartimento di Elettronica e Informazione
Politecnico di Milano, Milano, Italy
Email: baresi|guinea@elet.polimi.it

Marco Pistore Michele Trainotti
Service Oriented Applications Group
Fondazione Bruno Kessler
IRST, Trento, Italy
Email: pistore|mtrainotti@fbk.eu

***Abstract*—In the literature, there exist several approaches for monitoring the execution of BPEL processes. They concentrate on different properties, adopt different languages, work at different levels of abstraction, and assume different perspectives. Even if the field is rather new, we do not think that this diversity is a limitation of current solutions; rather it is intrinsic in the problem itself. We claim that, instead of working on the definition of the ultimate approach for BPEL monitoring, we should push a cooperative approach based on the integration of different solutions.**

In this paper, we present a first step in this direction, and describe a monitoring framework which is obtained by integrating two well-known approaches, namely Dynamo and Astro. This integration, which happens both for the language used for expressing the properties to be monitored, and for the architecture of the monitoring framework, allows to combine the advantages of the two approaches and to obtain a general, comprehensive solutions for BPEL monitoring.

I. Introduction

The conception of a software system as a composition of services imposes new requirements to validation techniques. Even if we think of BPEL (Business Process Execution Language [1]), a widely known technology for creating workflow-like assemblies of Web services, we must consider that service providers are entitled to change their services freely without notifying their users. This means that a service that originally was "correct" could become "incorrect" during its evolution. Besides this, a BPEL process is a highly distributed system where the entire computing power belongs to the partner services. The communication infrastructure may heavily impact the actual correctness of the composition: when it is too slow, or even broken, it may give the impression that services do not work, or they do not react with the required quality of service.

BPEL supports primitive forms of probing (e.g., timeouts) and exception handling, but these features do not cover all the different ways users would like to probe and control the execution of their processes. This is why there exist many —and quite diverse— proposals for monitoring the execution of BPEL processes. They concentrate on different properties, adopt different languages, work at different levels of abstraction, expose synchronous or asynchronous behaviours, and they can even assume different perspectives (e.g., the user is in charge of analysing the services it interacts with, or it is the provider that studies what it offers). Even the authors proposed their own monitoring solutions, with Dynamo [4], [5] defined for the synchronous check of properties on single process instances, and Astro [2], [3] that is more devoted to the specification of properties that span sets or classes of process instances. All these different characteristics clearly witness the importance of the problem, but also the impossibility for any single solution to take over the others.

Even if the field is rather new, we do not think that this is a limitation of current solutions; rather it is intrinsic in the problem itself. There are so many dimensions, and alternatives, that a single solution can hardly address all of them. We believe that interesting and complete solutions will come from the integration of different solutions rather than from augmenting existing ones. When a solution is good at something (e.g., checking temporal properties), it must keep its peculiarities; if it tries to embed too many things, the final result is often messy. The cooperation of different solutions helps get the positive aspects of the original proposals without compromising their efficiency.

In this paper we present a first experiment in this direction, and we describe a monitoring framework based on the integration of our two previous proposals Dynamo and Astro. The integration we propose is at two levels: at the *language* level and at the *architectural* level. The linguistic integration defines a language for specifying monitoring properties which layers the Astro language on top of the Dynamo language. The main challenge is to avoid conceptual redundancy, and to provide a solution that is as cohesive as possible. The architectural integration is carried out by means of a simple event bus that collects the relevant events on the process of interest and forwards them to the correct analysers. The main challenge here is to maximize code reuse, while at the same time accomodate a solution that is efficient. The paper demonstrates the high cohesion between the two approaches. Besides being the previous approaches conceived by the authors, Dynamo and Astro complement each other (see our analysis in [6]) and produce a novel and coherent solution for monitoring BPEL process executions. In this novel approach, the users can define constraints on single and multiple instances, on punctual properties and on complete behaviours, and they can also reason on both simple and composed properties.

The presentation is fully centered around a case study, which we use to demonstrate the key features of the novel framework. The case study in based on the process that guides the payment of the solid waste tax in Italian municipalities [13]. Besides this, we argue that the article is also interesting since it presents a general solution that can be easily generalised and applied to other approaches. In particular, the event bus helps decouple the process and its analyser(s), but it also allows us to fully control the distribution of the monitored events and data.

The rest of the paper is organised as follows. Section II introduces the case study used to illustrate the approach. Section III defines our integrated approach. Section IV details the implementation of our monitoring framework. Section V presents some of the most prominent related work in the field, while Section VI concludes the paper and presents some future work.

II. Scenario

In this paper, we consider the process that guides the payment of a specific municipal tax, the TA.R.S.U. (TAssa sui Rifiuti Solidi Urbani, i.e., municipal solid waste tax). The TA.R.S.U. is used to cover the costs of collecting and disposing solid waste, and is calculated on the basis of the size of the citizen's house, and of the number of people living in the house. The TA.R.S.U. process has been already used in [13] to experiment the adoption of advanced SOA technologies in a production context.

Figure 1 depicts the overall T.A.R.S.U. process. Due to lack of space, we have simplified it with respect to the real process. In particular, we only show the interactions the process has with the its partner services (i.e., its Receive and Invoke activities), while secondary BPEL activities (e.g., Assign) are ignored. Each block represents a BPEL activity with three different information: the type of activity (given in square brackets), the remote operation being called (or the operation on which the process receives a message), and the partnerlink on which the activity operates (given in angular brackets). In this scenario we have three remote partner services, and therefore three partnerlinks. UT represents the "Ufficio Tributi", i.e., the tax office, which is responsible of computing the amount of the taxes, as well as the penalties in case of late or missing payments of some citizen; UR represents the "Ufficio Ragioneria", i.e., the accounting office, which is responsible of registering the tax payments and, more in general, of managing the budget of the waste management procedures; and T represents the "Tesoriere", i.e., the treasury, which manages the collection of the taxes from the citizens.

The process is initiated by the Ufficio Tributi, which calculates the amount of taxes the citizen must pay, and sends a payment request to the TA.R.S.U. process ([RECEIVE] startRegistration<UT>). The process then sends an asynchronous registration request to the Ufficio Ragioneria ([INVOKE] requestRegistration<UR>). If the registration is unsuccessful ([ON MESSAGE] registrationFailure<UR>)), the Ufficio Tributi is notified about the failure and the process aborts ([INVOKE] registrationFailure<UT>). If the registration is successful (ON MESSAGE] registrationAck<UR>), the process notifies the Ufficio Tributi by sending it the registration data ([INVOKE] registrationStart<UT>), and routes the payment request to the Banca Tesoriere ([INVOKE] requestPayment<T>), which can answer with a success or a failure. If the request fails ([ON MESSAGE] paymentFailure<T>), the process notifies both the Ufficio Tributi ([INVOKE] paymentFailure<UT>) and the Ufficio Ragioneria ([INVOKE] paymentFailure<UR>), and terminates. If the request is successful ([ON MESSAGE] paymentAck<T>) the payment is notified to the Ufficio Tributi ([INVOKE] paymentNotification<UT>), and registered both with the Ufficio Tributi and the Ufficio Ragioneria ([INVOKE] registerPayment<UT> and [INVOKE] registerPayment<UR>). The registration with the Ufficio Ragioneria can fail, for instance if the due amount and the paid amount do not coincide. In this case ([MESSAGE ON] registrationFailure<UR>), the process notifies both the Ufficio Tributi and the Banca Tesoriere ([INVOKE] registrationFailure<UT> and [INVOKE] registrationFailure<T>). If the registration is successful ([INVOKE] registrationAck<UR>), the process sends an acknowledgement to the Ufficio Tributi ([INVOKE] registrationAck<UT>) and a receipt to the Banca Tesoriere ([INVOKE] accountantReceipt<T>).

A. Monitoring Requirements

While the TA.R.S.U. process is conceptually simple and generic, its implementation is far from trivial, as shown also in the above description. First it must account for a wide range of non-nominal cases: for instance, the citizen may not be able to fulfil his tax payment request; the payment may take too long to complete; or errors may arise in the transfer of personal data. All these failures are captured in the scenario above, and reported to Ufficio Ragioneria, Ufficio Tributi and Banca Tesoriere. This is however not sufficient: we need to monitor and collect information on these failures also at the level of the TA.R.S.U. process, in order to support a prompt analysis and reaction. Moreover, it is almost never the case that e-government processes handle citizens separately; for practical reasons, tax payment requests are usually handled in batches. For this reason, there is a need of collecting statistical information on the failures, but also on the performance of the system. We provide now some examples of monitoring properties that are important in the TA.R.S.U. domain.

1) We need to monitor how often the whole procedure fails, so to check that a reasonable success ratio is maintained at runtime. (From now on we will call this property *AverageProceduralFailure*).
2) We need to monitor the frequency with which the Ufficio Ragioneria is first notified of a citizen that needs to be recorded, but then the payment for that citizen fails. In this case effort is wasted, and it is important to monitor that such cases, while deemed possible, only take place seldomly. (Property *FrequencyPaymentFailure*).

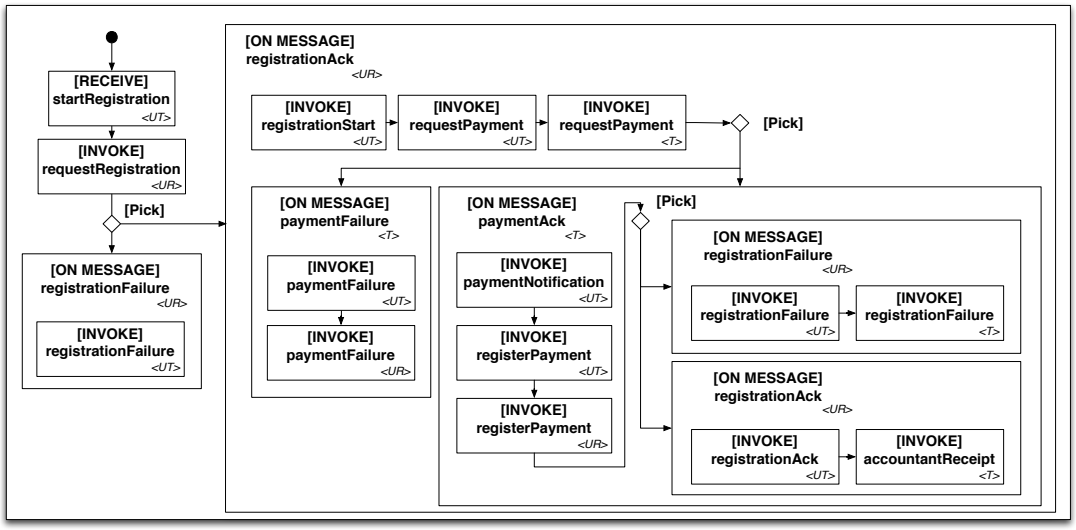

Fig. 1. TA.R.S.U. process.

3) We need to monitor the average time spent by the whole process. (Property *AverageTimeDuration*).
4) We need to monitor the average time spent by one of the actors in the process, e.g., the average time spent by the Banca Tesoriere for handling the payment procedure. This information is necessary to detect bottlenecks or deviations from estimated completion time. (Property *PaymentAverageTime*).

III. The Monitoring Approach

The approach presented in this paper was born from the conceptual fusion of two of our previous works: Dynamo and Astro. Dynamo [4], [5] uses Aspect Oriented Programming (AOP) [10] techniques to gather run-time data from a running BPEL process, and WSCoL (Web Service Constraint Language) to define the functional and non-functional properties that need to be checked during execution. Astro [2], [3], on the other hand, uses automatically generated and independent software modules to check properties that are defined in RTML (Run-Time Monitor specification Language).

This work is based on the comparison, undertaken in [6], of the expressive power of the two approaches. The goal of the fusion is to exploit the main advantages of both works, and to minimize their weaknesses by combining the expressive power of WSCoL and RTML. The result is characterized by the following main features. (1) It is possible to define fine-grained monitoring specifications thanks to the support for what we call "basic events". (2) The approach supports the collection of both internal and external data, allowing monitoring to consider context information when needed. (3) It is possible to define rich "composite" properties that take advantage of a Linear Temporal Logic. (4) A low degree of performance overhead is guaranteed. Although basic events are collected using AOP techniques, the actual analysis is performed by independent software modules that run in parallel to the process execution. This means that growing complexity in our monitoring specifications does not lead to higher performance costs. (5) The approach distinguishes between *instance* and *class* monitors. The former are monitors that check properties that are specific to a single BPEL process execution (or instance). The latter consider all the process executions that belong to the same process family (or class), and are particularly useful in the computation of statistics. (6) Monitors are automatically generated and deployed starting from a declarative definition. The code generation produces a state-transition system that evolves on the basis of events collected at run time.

A. Basic Events

Basic events are the data that we collect at runtime, and are used to evolve our monitor's state. The definition of basic events is heavily inspired by WSCoL. One reason is that WSCoL ensures, by definition, a high level of granularity in data collection. WSCoL (through the Dynamo framework) collects data at the level of a single BPEL activity. Another reason is that WSCoL provides advanced data selection and computation mechanisms.

A basic event is defined by a *declaration*, a *location*, and a *property*. The *declaration* defines the event's name, its type (see below), and its parameters, i.e., the run-time data needed to execute the basic event's property. We support two kinds of parameters, and call them internal and external variables. Internal variables consists of data that are collected from within the process' state, i.e., from the BPEL variables that exist within the process. External variables, on the other hand, are used to collect data that are not part of the executing process, such as context information. The values of the external variables are collected interacting with remote services that provide a WSDL interface. The *location* consists of an XPATH expression that selects a unique BPEL activity in the process

/* Basic events */

$e_b ::=$ ... any *boolean* or *tick* WSCoL formula ...

$e_n ::=$ ... any *numeric* WSCoL formula ...

/* Instance-level boolean formulas */

$b ::= e_b \mid b\ Since\ b \mid Once\ b \mid n > n \mid !\,b \mid b \&\& b \mid ...$

/* Instance-level numeric formulas */

$n ::= e_n \mid count(b) \mid time(b) \mid n + n \mid n * n \mid ...$

/* Class-level boolean formulas */

$B ::= And(b) \mid Once\ B \mid B\ Since\ B \mid N{>}N \mid !B \mid B \&\& B \mid ...$

/* Class-level numeric formulas */

$N ::= Count(b) \mid Sum(n) \mid Avg(n) \mid N + N \mid N * N \mid ...$

Fig. 2. The WSCol and RTML grammars.

definition. This is the location after which the basic event will be created. Finally, the *property* is given in an extended version of the WSCoL language. A property can produce one of three possible types: *boolean*, *numeric*, or *tick*.

Boolean properties are the typical WSCoL properties. Indeed in Dynamo WSCoL was used to produce truth values regarding a process' functional or non-functional property. Here the boolean values become the content of the basic event. Numeric properties can either produce a numeric datum using appropriate selection mechanisms on a complex BPEL variable, or using one of WSCoL's computation mechanisms. Finally, tick properties are used to express the fact that a given event has occurred. This is a new extension of WSCoL and allows us to produce a "tick" when a certain boolean property holds in a given location. When we define a tick event we express two possible payloads: the payload that will be attached if the property holds, and the payload that will be attached if it does not. A payload can be a boolean value or a number. We also provide a special keyword for stating that there is an empty tick (*TICK*), and a keyword for stating that no tick should be sent (*NOTICK*). Ticks also give us an easy way to signal that the process has reached a given location.

B. Composite Properties

Composite properties, expressed in RTML, take basic events and aggregate them into more complex properties. Notice that even through events are collected at specific locations in a process execution, composite properties are defined at the process level. More precisely, RTML distinguishes between *instance* and *class* monitor properties. Instance properties aggregate events from the execution of a single process instance (e.g., a single execution of the TA.R.S.U process). Class properties aggregate events from the executions of all process instance (e.g., a statistical property on all the executions of the TA.R.S.U. process).

The language allows to obtain both logical and quantitative information, and is based upon Linear Temporal Logic (see Figure 2 for the integrated language's grammar). Logical information is obtained using the standard logical operators such as && (and), || (or), and ! (not), as well as those typical of Linear Temporal Logic. With RTML it is possible to define properties over the whole past history of collected events, using formulae such as $f_1 Since f_2$ (formula f_1 has been true since the last instant formula f_2 has been true) or $Once f$ (f has been true at least once in the past). Numeric information is obtained using built in functions such as $count$ and $time$. Operator $count$ is used to count the occurrences of an event. Since RTML closely integrates logical formulae and numerical functions, we can use $count$ to count the occurrences of a complex behavior, or compare the values of multiple $count$ operators using boolean and relational operators. Operator $time$ measures the time-span for which a given boolean property stays true. We can use $time$ to calculate the time-span between two events e_{start} and e_{end} by means of formula $time(!\,e_{end}\ Since\ e_{start})$, which measures the duration of the interval starting whith event e_{start} and ending when $!\,e_{end}$ becomes false. We can also compare time-spans using relational and boolean operators. In addition, RTML provides the standard numerical functions ($+$, $*$, and so on), as well as comparison functions ($=$, $>$, and so on).

When defining a class monitor, RTML also provides additional class functions, such as And, $Count$, Avg, or Sum. These functions aggregate data over many different process instances. And, for instance, monitors whether a property is true for all instances of a process. $Count$ allows to count how many times a certain event occurs in all the process instances belonging to the same process class. Similarly, function Avg and Sum calculate the average and the sum of a numeric property evaluated across the different process instances.

C. Complete Examples

We can now provide complete specifications for the example monitoring properties given in Section II-A. All four properties define class monitors.

AverageProceduralFailure looks at the success ratio of the entire process, and can be defined as:

Basic events:

$$e_{fail} = tick((\$receipt/paymentID).length() == 16\ \&\&\ (\$receipt/paymentID).startsWith(\ (\$input/lastname).substring(1,3)+\ (\$input/firstname).substring(1,3))\ ?\ 0\ :\ 1)$$

Composite property:

$$Avg(e_{fail}?1:0)$$

In this case we associate process success with an analysis of the content of the final receipt. We state that the receipt must contain a payment identifier that is 16 characters long, and that the identifier must start with a sub-string that is the concatenation of the first three characters of the client's last name and the first three characters of the client's first name. The tick event is evaluated at location `[INVOKE] accountantReceipt`.

FrequencyPaymentFailure looks at the frequency with which a payment failure occurs, and can be defined as:

Basic events:
$e_{payfail} = tick((\$paymentAck/acceptedAmount) == (\$requestPayment/requestedAmount)\ ?\ 0\ :\ 1)$
Composite property:
$Avg(e_{payfail}?1:0)$

In this case we define the payment failure as an incongruency between the amount requested and the amount acknowledged by the Tesoriere. Other definitions are also possible. The tick event is evaluated at location `[ON MESSAGE] paymentAck`.

AverageTimeDuration looks at the average response time for the whole TA.R.S.U. process, and can be defined as:

Basic events:
$e_{start} = tick(true\ ?\ TICK\ :\ NOTICK)$
$e_{end} = tick(true\ ?\ TICK\ :\ NOTICK)$
Composite property:
$Avg(time(!\, e_{end}\ Since\ e_{start}))$

In this case we produce two tick events: one signaling the beginning of the process, and one signaling its correct completion. The two tick events are obviously evaluated at two different locations in the process. e_{start} is evaluated at location `[RECEIVE] startRegistration`, while e_{end} is evaluated at location `[INVOKE] accountantReceipt`.

PaymentAverageTime looks at the average time spent by the Ufficio Ragioneria in the payment process, and can be defined as:

Basic events:
$e_{startpay} = tick(true\ ?\ TICK\ :\ NOTICK)$
$e_{endpay} = tick(true\ ?\ TICK\ :\ NOTICK)$
Composite property:
$Avg(time(!\, e_{endpay}\ Since\ e_{startpay}))$

In this case we also produce two tick events: one signaling that we have just completed our request to the Ufficio Ragioneria to register the payment, and one signaling we have just received the Ufficio Ragioneria's response. The two tick events are obviously evaluated at two different locations in the process. $e_{startpay}$ is evaluated at location `[INVOKE] registerPayment`, while e_{endpay} is evaluated at location `[ON MESSAGE] registrationAck`.

These properties illustrate the advantages of combining Dynamo and Astro. On the one hand, Dynamo provides the granularity needed to predicate on very specific run-time data. For example, in **AverageProceduralFailure** Dynamo's data computation functions `startsWith` and `substring` allow us to predicate on the receipt's paymentID and on the user's first and last name. This level of granularity would have been impossible solely with Astro, where basic events consist of message exchanges. On the other hand, Astro provides the mechanisms needed to define class properties. For example, **AverageProceduralFailure** uses Astro's `Avg` function. Cross instance monitoring is potentially feasible with Dynamo, although highly inefficient. First of all the approach is synchronous with respect to process execution. Second, in this particular example, historical data would need to be collected and obtained as an external variable every time a new instance of the process is run. Astro also provides the functions we need to predicate on the time that passes between two basic events (function `time` in **AverageTimeDuration**).

IV. Architecture

The integrated framework revolves around ActiveBPEL, a well-known open-source BPEL implementation. It consists of two main components: the *ActiveBPEL Admin Console* and the actual *ActiveBPEL engine*. Both are web applications deployed in a standard tomcat container. The former allows managers to see what processes have been deployed, deploy new ones, and in general keep track of active processes. The latter, on the other hand, is responsible for executing the processes.

Along side we have added three additional components: the *Basic Events Manager (BEM)*, the *Composite Properties Monitor (CPM)*, and the *Monitoring Admin Console (MAC)*. The *BEM* is responsible for gathering basic events from executing processes, and for forwarding them to the *CPM*. The *CPM* is the container for our *Instance Monitor* and *Class Monitor* instances. Finally, the *MAC* is a web application where managers can see the results of the monitoring activities.

A. Basic Events Manager

The *Basic Events Manager* is the main gateway between the ActiveBPEL execution environment and the rest of the monitoring framework. It is implemented using aspect-oriented technology (i.e., AspectJ [9]), to guarantee an appropriate degree of separation between ActiveBPEL and our own components, and as a consequence between a process' business logic and its monitoring specifications. This makes it possible to change the amount of monitoring we perform, depending on the system's lifecycle or on context information, without having to touch the definition of the business logic. Such a clear separation is also important from an architectural standpoint, since it allows us to more easily keep up with the continuous evolution of ActiveBPEL's source code.

To fully understand how the *BEM* works, we must first briefly explain how the ActiveBPEL engine works. Once a process is deployed to the execution environment, ActiveBPEL creates a tree-like representation of the process' structure (i.e., it creates one node per BPEL activity in the process). We call these trees "definition" trees, since they contain the definition/configuration of the activities in the process. Every time the engine receives a message requiring the creation of a new process instance, the "definition" tree is used to instantiate what we call an "implementation" tree. Although there is always only one definition tree per deployed process, we have a new "implementation" tree every time we instantiate a process. Each implementation tree has access to the process instance's unique internal state, i.e., to the values associated with the BPEL variables that exist in the process. This separation allows

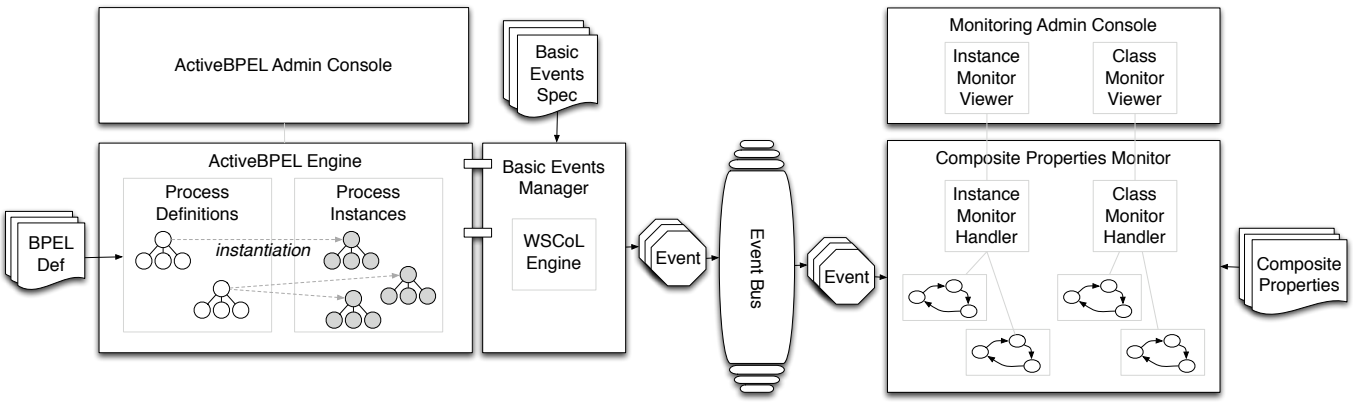

Fig. 3. Architecture of the integrated approach.

the system to minimize the amount of information that needs to be replicated across "implementation" trees.

During execution, the engine takes advantage of the fact that every node in the implementation tree implements a common Java interface called `AeActivityImpl`. This means that all the nodes in the tree implement a method called `execute`. Therefore, the engine simply traverses the implementation tree and calls method `execute` on each of the nodes it encounters. During the traversal, we use AOP to activate the code for creating basic events. In practice, we activate additional code after the engine calls method `execute` on nodes type `AeActivityReceiveImpl` (for BPEL Receive activities) or `AeActivityReplyImpl` (for BPEL Reply activities). We also activate our AOP code when the engine performs BPEL Invoke activities. However, to support both synchronous and asynchronous invocations, ActiveBPEL implements Invoke activities in a slightly different manner. This means we need to add our code before `AeActivityInvokeImpl` completes the execution of method `objectCompleted`. We also inject code after the process is instantiated, and before the process instance is terminated. These are important points of execution for treating the process' lifecycle.

The code injected into the normal execution gathers the internal and external data that need to be packaged into a basic event. In the first case, the data is obtained by accessing the process' internal state using ActiveBPEL's own APIs. In the second case, we use JAX-WS technology to call an external Web service and gather the information we need. Since the definition of a basic event contains a property, the *BEM* contains a *WSCoL Engine*. Before sending off the event, we also add meta-level information, such as the process instance ID, the process' name, the partnerLink on which the interaction with the outside world is occurring (if possible), and the name of the operation being called. Notice that most of these information are retrieved from the "definition" tree. When the event is ready, it is sent to the *CPM* and the process instance is allowed to continue its execution.

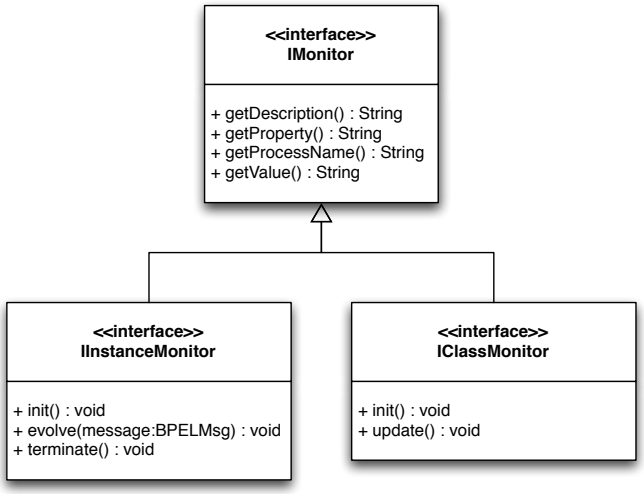

Fig. 4. Monitor Interfaces.

B. Composite Properties Monitor

The *Composite Properties Monitor* is a container for monitor instances. Inside the container we have a *Instance Monitor Handler* and a *Class Monitor Handler*, in accordance with the two kinds of monitors supported in our approach. The handlers are responsible for managing both a monitor's lifecycle (creation and termination) and its evolution. Each monitor is a Java software module that is run in parallel to its corresponding BPEL process, and is automatically generated from a composite property. Each monitor implements the `IMonitor` interface described in Figure 4. More precisely, instance monitors implement interface `IInstanceMonitor`, while class monitors implement interface `IClassMonitor`.

The methods in `IMonitor` are for gathering general information from a monitor instance. `getProperty` and `getDescription` return the composite property being checked and a long description of the monitor. `getProcessName` returns the name of the process the monitor is associated to, and `getValue` gives the mon-

itor's current (truth or numerical) value. The methods in `IInstanceMonitor` and `IClassMonitor`, on the other hand, are used to evolve a monitor's state.

The lifecycle of an instance monitor is affected by three kinds of events: the process creation event, basic events from the *BEM*, and the process termination event. When the *CPM* receives a process creation event it forwards it to the *Instance Monitor Handler*, which creates all the instance monitors that need to be associated with the new process instance. This is achieved instantiating new `IInstanceMonitors`, and calling method `init` on each one. When the *CPM* receives a basic event it forwards it to the *Instance Monitor Handler*, which in turn uses it to call method `evolve` on the right instance monitor[1], causing the monitor to update its truth value. Finally, the arrival of a termination event causes the *Instance Monitor Handler* to call method `terminate` on the corresponding instance monitors, allowing them to gracefully terminate.

The lifecycle of a class monitor is different. The *Class Monitor Handler* only calls method `init` the first time a process instance is created. This is because all subsequent process instances that belong to the same process class are associated to the same class monitors. Indeed, the class monitors evolve every time a basic event for that process class is received, regardless of the process instance that generated it. When a basic event is received by the *CPM*, it is passed to the *Class Monitor Handler*, which calls method `update` on the corresponding class monitors. Class monitors can also interact with the running instance monitors that are associated with processes of the same class. This is useful when a class monitor is responsible for collecting statistical data on all the instances of a given BPEL process.

C. Monitoring Admin Console

The *Monitoring Admin Console* (see Figure 5 is a web application where process managers can see how their monitoring activities are proceeding. The console consists of two main components: the *Instance Monitor Viewer* and the *Class Monitor Viewer*. These components interact with the corresponding handlers within the *CPM* to discover the current status of all the running monitors. The viewers provide managers with a high-level representation of the monitor truth values, as well as with the low level details that are contained within the logs created by the monitors.

V. Related Work

In the field of service monitoring, many works concentrate on BPEL processes. However there are also more general approaches, e.g. those that revolve around the concept of Service Level Agreement (SLA), or those that revolve around WS-Policy. Due to lack of space we provide a short presentation of some of these approaches and classify them. More details can be found in literature.

[1]The right instance monitor is chosen based on the arriving event's *Process Instance ID*.

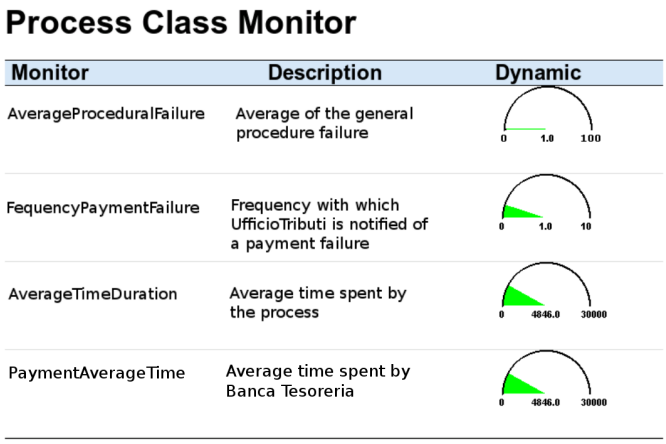

Fig. 5. Monitoring Admin Console.

Regarding BPEL-based approaches, Spanoudakis et al. [11] propose a framework for validating behavioural properties expressed using the first order temporal logic language of event calculus. The events are the exchange of messages between the process and the systems it coordinates. The approach is non-invasive, and monitoring is performed independently from the process' execution. Moser et al. [12] present VieDAME, a non-intrusive approach based on AOP techniques. It is pluggable with respect to different engines. It concentrates on the evaluation and monitoring of simple QoS values such as response time, accuracy, and availability. Complex properties are not tackled in their approach.

Regarding more general approaches that revolve around SLAs, we cite Skene et al.'s work on SLANG (language for SLAs) [14]. The approach mitigates risk by associating penalties to anomalous behaviours, and by using meta-modelling techniques clarify the terms of an SLA. This allows different partied to defend their interests in the wake of critical problems.

Finally, regarding approaches that deal with WS-Policy, we cite Erradi et al.'s work on WS-Policy4MASC [8]. They provide a general-purpose language for defining policies within WS-Policy. Designers can define the source of the monitoring data (both internal and external to the service) and the modality of monitoring (synchronous or asynchronous). The authors provide a .NET implementation that requires the services and process be run in the Microsoft Workflow Foundation toolset.

To conclude we present a table that compares the approaches we have cited (see Table I). We include Dynamo with WSCoL and Astro with RTML, as well as the integrated approach presented in this paper. The table follows the taxonomy presented by Delgado et al. [7], with some modifications/extensions of the metrics to adapt them to the service-oriented context.

VI. Conclusion and Future Work

In this paper, we presented a novel approach for monitoring BPEL orchestrations, which has been obtained by the integration of two previous approaches conceived by the authors,

TABLE I
COMPARISON OF MONITORING APPROACHES.

Approach	Language		Abstraction		Properties		Directives			Timeliness	
	Logic	HL/VHL	Domain	Implementation	Safety	Temporal	Process	Activity	Event	Synchronous	Asynchronous
Spanoudakis et al.	x			x	x	x			x		x
Moser et al.		x		x	x			x			x
Skene et al.		x	x		x		x				x
Erradi et al.	x	x		x	x	x		x		x	x
Baresi et al.	x			x	x			x		x	
Pistore et al.	x			x	x	x	x		x		x
Our Approach	x			x	x	x	x	x	x		x

namely Dynamo and Astro. The integration is both at the language level —the new language combines in a suitable way the WSCol specifications of Dynamo and the RTML specifications of Astro— and at the architectural level —an event bus is exploited to route events collected by Dynamo to the property monitors implemented by Astro.

Although the linguistic integration is suitable, we believe it can benefit from higher level abstractions or templates that simplify the definition of complex properties (e.g., statistical properties). The architectural integration seems to place the foundation for a general solution that can be applied to other approaches. In particular, the event bus allows for the integration of more monitoring engines (even non open-source) that can collaborate in order to check complex, multi-layered monitoring properties. To do so we have to separate the definition of basic events with WSCoL from RTML in a sharper manner, and allow basic events to be used by any monitoring engine that is registered on the bus. We can avoid performing integration at the linguistical level, and formalize the nature of the basic events that are generated by the infrastructure. In this case, the integration is performed solely at the architectural level. In our future work, we intend to push this idea, and to integrate other monitoring approaches within our framework. Another line of future investigation is to go beyond monitor, and to also consider reactions to the deviations recognised by the monitors. Our intuition is that a modular approach based on the collaboration of various adaptation engines can help also in this case, and we are working on the architecture of such an adaptation framework and in understanding and defining the role of the "adaptation bus" in this context.

ACKNOWLEDGMENT

The research leading to these results has received funding from the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreements 215483 (S-Cube Network of Excellence) and 216556 (SLA@SOI Project).

REFERENCES

[1] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business Process Execution Language for Web Services, Version 1.1. BPEL4WS specification, May 2003.

[2] F. Barbon, P. Traverso, M. Pistore, and M. Trainotti. Run-time monitoring of instances and classes of web service compositions. In *Proc. ICWS'06*, pages 63–71. IEEE Computer Society, 2006.

[3] F. Barbon, P. Traverso, M. Pistore, and M. Trainotti. Run-time monitoring of the execution of plans for web service composition. In *Proc. ICAPS'06*, pages 346–349. AAAI, 2006.

[4] L. Baresi and S. Guinea. Towards dynamic monitoring of ws-bpel processes. In *Proc. ICSOC'05*, LNCS 3826,pages 269–282. Springer, 2005.

[5] L. Baresi and S. Guinea. A dynamic and reactive approach to the supervision of BPEL processes. In *Proc. of the 1st conference on India software engineering conference*, pages 39–48. ACM New York, NY, USA, 2008.

[6] L. Baresi, S. Guinea, R. Kazhamiakin, and M. Pistore. An integrated approach for the run-time monitoring of bpel orchestrations. In *Proc. ServiceWave 2008*, pages 1–12, 2008.

[7] N. Delgado, A. Q. Gates, and S. Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Transactions on Software Engineering*, 30(12):859–872, 2004.

[8] A. Erradi, P. Maheshwari, and V. Tosic. WS-policy based monitoring of composite web services. In *Proc. ECOWS'07*, pages 99–108. IEEE Computer Society, 2007.

[9] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An Overview of AspectJ. In *Proc. of the 15th European Conference on Object-Oriented Programming*, LNCS 2072, pages 327–353. Springer, 2001.

[10] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proc. ECOOP'97*, LNCS 1241, pages 220–242. Springer, 1997.

[11] K. Mahbub and G. Spanoudakis. A framework for requirents monitoring of service based systems. In *Proc. ICSOC'04*, pages 84–93. ACM, 2004.

[12] O. Moser, F. Rosenberg, and S. Dustdar. Non-intrusive monitoring and service adaptation for ws-bpel. In *Proc. WWW'08*, pages 815–824. ACM, 2008.

[13] M. Pistore, P. Braghieri, P. Bertoli, A. Biscaglia, A. Marconi, S. Pintarelli, and M. Trainotti. ASTRO: Supporting the composition of distributed business proce sses in the e-government domain. In *At Your Service - Service-Oriented Computing from an EU Perspective*, pages 183–211. MIT Press, 2009. To appear.

[14] J. Skene, D. Lamanna, and W. Emmerich. Precise Service Level Agreements. In *Proc. ICSE'04*, pages 179–188, 2004.