An Integrated Approach for the Run-Time Monitoring of BPEL Orchestrations*

Luciano Baresi[1], Sam Guinea[1], Raman Kazhamiakin[2], and Marco Pistore[2]

[1] Politecnico di Milano – Dipartimento di Elettronica e Informazione, Italy
{baresi,guinea}@elet.polimi.it
[2] Fondazione Bruno Kessler – IRST, Trento, Italy
{raman,pistore}@fbk.eu

Abstract. In this paper, we compare and integrate Dynamo and ASTRO, two previous approaches of the authors for run-time monitoring of BPEL orchestrations. A key element of the proposed integrated framework is the capability to cover a wide range of features, including the detection of complex behavioural patterns, the possibility to measure boolean, numeric and time-related properties, the possibility to monitor the behaviour of the composition both at the level of a single execution instance and by aggregating the information of all execution instances of a given composition.

1 Introduction

BPEL (Business Process Execution Language) is the most widely used solution for workflow based cooperation amongst web services. The distributed nature of BPEL processes, the absence of a single stakeholder, the fact that partner services can dynamically change their functionality and/or QoS, and the possibility to define abstract processes and look for actual services at run time, preclude design- time validation of such systems. The reliability and robustness of these systems must be enforced by means of defensive programming techniques and suitable monitoring of executions, in order to detect problems and trigger recovery activities. BPEL supports primitive forms of probing (e.g., timeouts) and exception handling, but these features are not as powerful and flexible as needed. For this reason, several approaches have been proposed in literature for specifying monitoring directives externally to the BPEL processes and for supporting the run-time monitoring of these directives – see for instance [2,3,4,5,6,7,8,9,10,11]. Among all these different alternatives, we focus here on two approaches developed by the authors of this paper, namely Dynamo [5,6] and ASTRO [2,3]. Even if both approaches address the problem of the run-time monitoring of BPEL processes, the developed solutions are rather different. Indeed, in [5,6] the focus is on the specification of monitoring directives that can be activated and de-activated for each process execution, according to the user's preferences; the actual monitoring of these directives is performed by weaving them into the process they belong to. In [2,3], instead, the focus is on the specification of properties which may span over multiple executions of BPEL

* The research leading to these results has received funding from the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement 215483 (S-Cube).

P. Mähönen, K. Pohl, and T. Priol (Eds.): ServiceWave 2008, LNCS 5377, pp. 1–12, 2008.

processes and that aggregate information about all these executions; moreover the architecture clearly separates the BPEL execution engine and the monitoring engine.

In this paper, we perform a detailed comparison of the two approaches, in terms of the basic events they are able to monitor, the way these basic events can be combined to monitor complex properties, the granularity of executions that a monitor can cover (single execution vs multiple executions), and the level of integration of process execution and monitoring (e.g., tangled together or separated). The outcome of this comparison is that the two approaches have taken complementary approaches in all these perspectives, and as a consequence they have complementary strengths and weaknesses. This complementarity opens up the possibility of combining the two approaches in order to exploit the strengths of both of them. In this paper, we propose a novel approach that is obtained by integrating the two existing approaches. We describe both the language and the monitoring architecture for this new approach. A key element of the new framework is the capability to cover a wider range of features than any other monitoring approach for BPEL orchestration the authors are aware of.

The organisation of the paper is as follows. Section 2 presents a simple case study used throughout the paper. Section 3 and 4 briefly introduce Dynamo and ASTRO, while Section 5 compares them. Section 6 sketches the integrated approach resulting from combining the two aforementioned proposals. Section 7 concludes the paper.

2 Tele-Assistance Service

The Tele-Assistance Service (from now on `TA`) is a BPEL process that manages the remote tele-assistance of patients with diabetes mellitus. These patients have glucose meters at home that communicates with `TA`, allowing their glucose levels to be constantly monitored. Figure 1 illustrates the overall process. It uses square brackets to indicate the nature of the BPEL activity, and angular brackets to indicate the remote partner service being called. The process interacts with five partners: (a) the patient's home device (`PHD`), (b) the hospital's patient record registry (`PRR`), (c) the hospital's medical laboratory (`LAB`), (d) a pool of doctors that can provide on-the-fly home assistance (`DOC`), and (e) an ambulance emergency center (`AMB`).

A new instance of service `TA` is instantiated remotely when the client turns on his/her glucometer, which sends an appropriate `startAssistance` message. The process starts by obtaining the patient's medical records from `PRR`, in order to tell the patient (through their glucometer) the exact insulin dose to use. Once this "setup" phase is concluded, `TA` enters a loop in which a BPEL pick activity is used to gain information from the outside world. The pick activity defines three possible `[ON MESSAGE]` branches. Notice that all the decisions taken within the process are made persistent by calling the `PRR` service. The first branch, called `vitalParams`, is used to receive a periodic update from the patient's glucometer. It immediately starts by sending the data to the medical lab (`[INVOKE] analyzeData`) for an online analysis; when the results are ready they are used to decide what to do. If the patient's results are fine, no action is performed. If there are reasons to change the insulin dose being used, this is communicated to the patient (`[INVOKE] changeDose`). If the results highlight a "mild" anomaly, it is communicated to the doctors (`[INVOKE] alarm('mild')`) so that they can schedule a visit

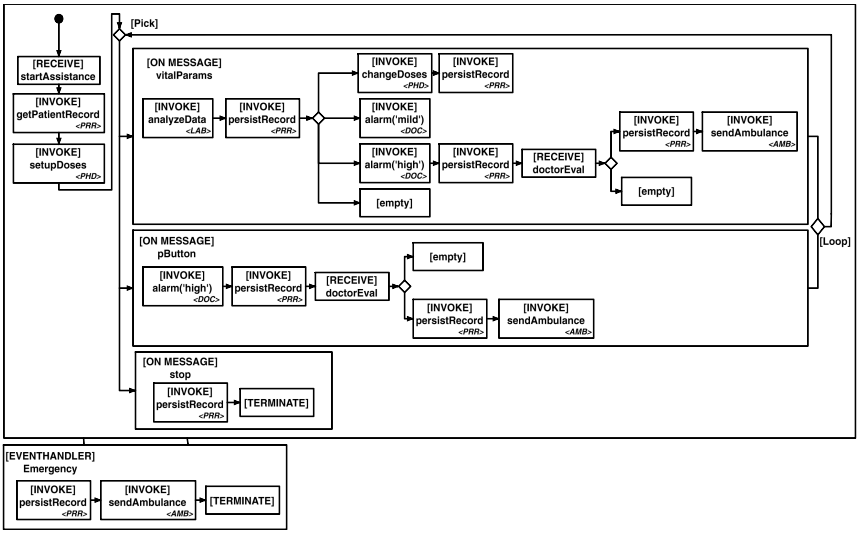

Fig. 1. The Tele-Assistance Service

to the patient. Notice that whenever the process contacts the doctors it expects an immediate acknowledgement message. If the results highlight a "serious" anomaly, it is communicated to the doctors (`[INVOKE] alarm('high')`). This causes the doctors to rush to the patient's house and perform a complete checkup. The results are received by `TA` through a new receive activity called `doctorEval`, after which two things can happen. Either the doctors have solved the problem and the `TA` service can continue regularly, or the problem cannot be solved easily and hospitalisation is deemed necessary. In this case an order is sent to the ambulance (`[INVOKE] sendAmbulance`) so that it can go pick up the patient. The second branch, called `pButton`, represents a panic button that the patient can use to communicate an emergency. This emergency is treated exactly as in the previous thread. The third branch, called `stop`, can be used to terminate the `TA` service. This branch is called either when the patient turns off his/her glucometer, or when the ambulance picks up the patient for hospitalisation. The overall process also provides an `[EVENTHANDLER]`, which can be used at any time to require the patient's immediate hospitalisation. It is used by the doctors, for example, to require hospitalisation in case of a "mild" alarm which turns out to be worse than expected.

In the following, we report some examples of monitoring directives that are relevant in this service. A *first property* is that the changes in the insulin doses suggested by the medical lab analysis should not vary "too" much during a short period. For instance, we may require that a suggested insulin dose should not differ from the previous value by more than 5%. Or we may require that the difference between the highest and the lowest suggested doses are within a range of 20%. A *second property* is on the performance of the service. For instance, we may want to monitor that the time it takes the doctors to send back an acknowledgement must never exceed 500ms, and that the average response time should be below 200ms. Notice that the average response time should be computed considering all the executions of the `TA` service, not just one. A *third property*

is on the number of times a given event occurs: for instance, we may be interested in knowing the number of times hospitalisation has been necessary for a given patient. Notice that this property should be computed considering all the executions of the TA service for a given patient. A *fourth property* is on the temporal behaviour of the service: for instance, we may be interested in monitoring whether hospitalisation has been decided after the insulin dose has been progressively incremented for three or more times; we may also be interested in knowing the percentage of cases where such an increment in the dose has lead to hospitalisation. *Other properties* may result from the combination of the TA process described in this section with other services. For instance, if a patient is monitored with more sensors than just the glucometer (e.g., sensors for heart rate, pression, temperature...) and these sensors are managed by other services, then would be important to define monitoring properties that correlate conditions and events across these services.

3 Dynamo

In Dynamo [5,6] monitoring rules are made up of a location and a monitoring property. Additionally, a set of reaction strategies can be defined [6], but this is out of the scope of this paper. The *location* uses an XPath expression to select the point in the process for which we are defining monitoring, and a keyword for the "kind" of triggering condition we want (a pre- or a post-condition). Possible points of interest are for instance BPEL invoke and receive activities.

Monitoring *properties* are defined using WSCoL, an XML-aware language for the definition of behavioural properties. WSCoL defines three kinds of variables (i.e., internal, external, and historical), and expresses relationships that must hold between them. Internal variables consist of data that belong to the state of an executing process, and are defined by indicating the name of a BPEL variable (preceded by a `$`) and an XPATH expression that "chooses" one of the simple data values it contains (i.e., a number, a string, or a boolean). External variables consist of data that cannot be obtained from within the process, but must be obtained externally through a WSDL interface. This solution facilitates the distribution of probes and helps control the deployment of the monitoring infrastructure. This also allows for specifying certain QoS properties that can only be collected with the help of special purpose probes. Finally, historical variables are introduced to predicate on internal and external variables collected during previous activations of the monitoring framework, either from within the same process execution or from a completely different process.

In WSCoL, we can also define variable aliases. This allows us to write simpler and clearer properties, and more importantly, when we have the same external variable referenced more than once in a property, we can either collect it is as many times as needed or collect it once and define an alias for future references. This is crucial when the value of an external variable may vary depending on the exact moment in which it is collected. To define the relationships that must hold among these variables, we can use the typical boolean, relational, and mathematical operators. The language also allows us to predicate on sets of values through the use of universal and existential quantifiers, and provides a number of aggregate constructs such as `max`, `min`, `avg`, `sum`, `product`,

and num_Of. These constructs become quite meaningful in conjunction with historical variables and allow us to compare the behaviour of a remote service with previous iterations. To better clarify how WSCoL is used, we present the following monitoring properties, defined in the context of our TA service example.

```
let $doseNew=($labResults/suggestedDose);
let $doseOld=retrieve(pID, uID, iID,
      '[INVOKE]changeDose/postcondition', '$doseStored', 1);
$doseNew <= $doseOld*1.05 && $doseNew >= $doseOld*0.95;
```

In this example we define a post-condition for [INVOKE] analyzeData, in which we state that the change in the insulin dose suggested by the medical lab analysis should not differ from the previous value by more than 5%. The property uses two variable aliases. The former (i.e., $doseNew) is defined for an internal variable responsible for extracting the new insulin dose from the BPEL variable labResults using the XPath expression /suggestedDose. The latter (i.e., $doseOld) is defined for a historical variable. The historical variable is obtained using the appropriate WSCoL retrieve function. The function takes, as parameters, the process name, the user ID, and the instance ID, allowing us to indicate that we are only interested in variables that were stored from within the process instance in execution. Its remaining parameters, on the other hand, allow us to state that we are interested in a variable that was stored in [INVOKE] changeDose's post-condition, and that was called "doseStored".

This example can be extended by stating that the difference between the highest and the lowest dosage suggestions should be within a 20% range. To do this we must calculate the maximum and minimum of the last 10 dosages that were stored:

```
let $vals=retrieve(pID, uID, iID,
      '[INVOKE]changeDose/postcondition', '$doseStored', 10);
let $min= (min $d in $vals; $d); let $max= (max $d in $vals; $d);
$min > $max * 0.80;
```

Here we use the min and max aggregate functions, which return the minimum or maximum of a parametric expression calculated using values taken from a finite range. In this case the range is the set of dosages extracted from the historical storage ($stored), and the expression to calculate is the value itself ($d).

As a second example, we add a pre-condition to [INVOKE] alarm('high') stating that average time it takes the doctors to send back an acknowledgement must not exceed 200ms, and that a single invocation should never take more than 500ms.

```
let $range = retrieve(pID, null, null,
      '[INVOKE]alarm('high')/postcondition', $rt, 50);
(avg $t in $range; $t) < 200 && $rt < 500;
```

$rt is a special purpose keyword that can be used only in post-conditions and that refers to the amount of time it took the service to respond. $range, on the other hand, retrieves the last 50 $rts stored in [INVOKE]Alarm('high')'s post-condition. With respect to the previous example, we use the receive function to collect historical variables that belong to the entire process family. We are not looking at the response times that this

process instance has experienced, but at all the response times experienced by all the instances of service TA.

As a third example, we are interested on predicating on the number of times a patient is hospitalised. This needs to be computed considering all the executions of the TA service for a given patient. We will state that hospitalisations should be less than 3. We could use this, for example, to signal that a fourth hospitalisation should not be requested without contacting the patient's doctor directly.

```
let $hosps = retrieve(pID, uID, null, null, $hospEvent, 10);
(num_Of $h in $hosps; $h) < 3;
```

$hosps contains the last 10 $hospEvent variables added to the historical storage. The num_Of aggregate function provided by the WSCoL language allows us to count how many values in a range satisfy a given property. In this case, we use this function to count how many $hospEvents (aliased as $h) were extracted from the historical storage, and compare its value with the constant 3.

To store these values in the historical storage we need to use the WSCoL store function. The following WSCoL code can be added throughout the TA service, in all those points in which a hospitalisation is performed.

```
let $hospEvent = 1; store $hospEvent;
```

Notice that in our retrieve function we only specify the user ID, and the process ID, but neither the instance ID nor the location in which the $hospEvents were stored. This is a must since there are different place in the process in which the hospitalisation may have been requested. We are interested in all of them.

4 ASTRO

In ASTRO [2,3], the approach to monitoring is characterised by three main features. (1) Monitors are independent software modules that run in parallel to BPEL processes, observe their behaviour by intercepting the input/output messages that are received/sent by the processes, and signal some misbehaviour or, more in general, some situation or event of interest. That is, the ASTRO approach does not require the monitored services to be decorated or instrumented to any extent, in order to guarantee direct usage of third party services. (2) The approach supports two different kinds of monitors: *instance monitors*, which observe the execution of a single instance of a BPEL process; and *class monitors*, which report aggregated information on all the instances of a given BPEL process. The latter assume a crucial importance to enact the computation of statistics and that therefore aggregate multiple executions of other processes. (3) Monitors are automatically generated and deployed starting from properties defined in RTML (Run-Time Monitor specification Language), which is based on events and combines them exploiting past-time temporal logics and statistical functionalities.

RTML stands on the notion of monitorable event, defining what can be immediately tracked by a monitoring process which observes the behaviour of services. Basic events correspond message exchanges (for instance, msg(TA.output = changeDose)

denotes the emission of a recommendation of changing the insulin dose by the `TA` service), and creation and termination of service instances (denoted by the `start` and `end` keywords). Regarding *instance monitor* properties, RTML offers the ability to obtain monitor information of both logical and quantitative nature. The *logical* portion of RTML consists of standard boolean operators, and of a past-time linear temporal logic; that is, RTML allows for specifying properties on the whole past history of a given instance monitor, using "temporal" formulas 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). Such kinds of RTML expressions are useful to track down unexpected behaviours of a service, which in most cases can be represented by simple linear logic formulae. For instance, the following formula checks whether an hospitalisation is requested after at least one request to change the insuline dose:

```
msg(TA.output= sendAmbulance) && Once(msg(TA.output= changeDose))
```

The *numeric* portion of RTML allows counting events (operator `count`), and computing the time-span between events (operator `time`); this is very useful when checking, for instance, the QoS of the service instance being monitored. Indeed, the numeric and logical portions of RTML are closely integrated, so that it is possible to count the occurrences of complex behaviours (represented by temporal formulae), or vice versa, to trigger a boolean monitoring condition based on comparisons amongst numerical quantities (for instance, a certain event taking place more often than expected). An example of a numeric property is the fact that a hospitalisation is requested after three requests to change the insulin dose:

```
msg(TA.output = sendAmbulance) &&
  count(Once(msg(TA.output = changeDose))) = 3
```

An example of a time-based property is the fact that the time it takes the doctors to send back an acknowledgement (i.e., the time during which no `doctorEval` is received since the last `alarm ('high')` event) must never exceed 500ms:

```
time((!msg(TA.input = doctorEval)) Since
        (msg(TA.output = alarm('high'))) <= 500ms
```

Besides the instance monitors we considered so far, RTML also offers the possibility to specify *class monitors*, which aggregate monitoring results over all executions (or instances) of a given BPEL process. For instance, using an appropriate "class count" operator `Count`, it is possible to compute the total amount of times a certain property holds through all executions of a given process. For instance,

```
Count(msg(TA.output = sendAmbulance) &&
        count(Once(msg(TA.output = changeDose))) = 3)
```

computes the total number of times a hospitalisation has followed three recommendations to change the insulin dose. Similarly, an appropriate "class average" operator `Avg` allows us to compute the average times spent by services undertaking certain tasks. So,

```
Avg(time((!msg(TA.input = doctorEval)) Since
            msg(TA.output = alarm('high')))) <= 200ms
```

constrains the average time it takes to the doctors to answer to an emergency.

The ASTRO approach to support monitoring is based on converting RTML properties into state-transition systems that evolve on the basis of basic events of the process executions (e.g., reception and emission of messages). The states of the these state-transition systems codify the current evaluation of the formula, so that certain states are associated to the satisfaction of the monitoring requirement, while other states correspond to failures. A *monitoring engine* is responsible for receiving the relevant events from the BPEL execution engine, for correlating these events to the monitoring properties which depend on these events, for progressing the state of the state-transition systems that correspond to these properties, and for reporting failures and violations. This engine is built as an extension of ActiveBPEL [1], one of the most prominent engines for executing BPEL processes. The ASTRO extension allows for intercepting input/output messages and other relevant events such as the creation and termination of process instances. The extension also includes the ActiveBPEL admin console, which is exploited to report the information on the status of the monitors.

5 Comparison

In this section, we draw a comparison between Dynamo and ASTRO. This comparison will cover different aspects of a monitoring approach, namely: the kinds of basic events the approaches are able to monitor, the way these basic events can be combined in order to monitor more complex properties, the granularity of executions that a monitor can cover, and the level of integration of process execution and monitoring.

Basic events. We identify three different kind of basic events. *Messages*, i.e., the fact that a given message (with given values) is sent or received by the process; this kind of basic event can be managed by both approaches. *Control points*, i.e., the fact that the execution has reached a given point of the BPEL process; only Dynamo supports this kind of basic event. These events contain the variable values that are visible at that point in the process (in accordance with BPEL's own scoping rules). *Life cycle*, i.e., the possibility to monitor events related to the life cycle of a service execution, such as the fact that a new execution of a service is started, that the execution terminates with success or with an exception; only ASTRO supports this kind of basic events.

Event combination. We identify three different dimensions among which previous events can be combined in order to express complex monitoring properties. *Statistical* dimension, i.e., the possibility of producing aggregated information on a set of variables; both approaches support this dimension through operators such as `max`, `min`, `avg`, `sum`, `count`. *Time* dimension, i.e., the possibility of measuring durations and time intervals between events; this dimension is supported in a native way by ASTRO, while it can be encoded in Dynamo through external variables. *Temporal* (or behavioural) dimension, i.e., the possibility of expressing properties on the temporal evolution of the service, or behavioural patters that consist of sequences of events; this dimension is supported in a native way by ASTRO, through the use of temporal logic operators; it can be encoded in Dynamo through historical variables and quantifiers.

Granularity. This aspect is related to the granularity of process executions that are covered by a monitoring property. In particular, a property can refer to the following

execution granularities. *Location*, i.e., the property is associated with a specific location of a BPEL process, and the monitoring is performed when the execution of a process instance reaches that location; Dynamo is based on this kind of granularity. *Process instance*, i.e., the property is associated with a single execution instance of a BPEL process, and the monitoring is performed through the instance execution; both approaches support this natively. *Process class*, i.e., the property is associated with all the executions of a BPEL process; ASTRO supports this kind of granularity in a native way (through class monitors); in Dynamo, these properties can be monitored using historical variables, quantifiers, and aggregated operators. *Cross-process*, i.e., the monitoring property can correlate events that refer to (execution instances of) different processes; ASTRO does not support this granularity; Dynamo requires encoding them in terms of "local" monitors and additional state variables.

Integration level. This aspect refers to the level of integration of the process execution engine with the monitor execution language. We identified the following levels of integration. *At the level of the BPEL specification*, i.e., the monitor is performed by instrumenting the BPEL specification with specific instructions that activate and advance the monitoring engine; this is the level of integration supported by Dynamo. *At the level of the BPEL engine*, i.e., the BPEL engine is tightly integrated with the monitoring engine and their executions are synchronised; neither Dynamo nor ASTRO support this level of integration. *Through asynchronous events* sent by the BPEL engine (or any other event source) to the monitoring engine; the monitoring engine is responsible for deciding which events are relevant for which monitors, and for advancing the monitoring task; ASTRO adopts this level of integration.

Discussion. A summary of the comparison of the Dynamo and ASTRO approaches is reported in Table 1. (The last row corresponds to the integrated approach defined in Section 6). The comparison shows that the two approaches are complementary under several aspects. In particular, Dynamo defines monitoring properties that are attached to specific locations of the BPEL specification, while ASTRO defines monitors that are associated to the whole execution of a BPEL specification. This difference also has effects on other aspects: on the level of integration between service execution and monitoring (strictly interconnected in the case of Dynamo, mediated by events in the case of ASTRO); on the capability to access the internal state of the BPEL process (supported by Dynamo but not by ASTRO); and on the capability of expressing properties that combine events in a complex way (easy to achieve in ASTRO, while slightly more difficult in Dynamo).

Table 1. Comparison of Dynamo and ASTRO approaches

	Basic events			Combination			Granularity				Integration level		
	mes-sages	ctrl points	life cycle	statis-tical	time	tem-poral	loca-tion	in-stance	class	cross process	BPEL	engine	events
Dynamo	+	+	–	+	+/–	–/+	+	+	–/+	–/+	+	–	–
ASTRO	+	–	+	+	+	+	–	+	+	–	–	–	+
Int.Appr.	+	+	–	+	+	+	+	+	+	+	+	+	+

Legenda: +: yes –: no +/–: yes, with light encoding –/+: yes, with heavy encoding

6 Integration

In this section we discuss a possible way of integrating the approaches of Dynamo and ASTRO, trying to exploit as much as possible the complementarity of the two approaches and to achieve the highest level of expressiveness.

Basic events. We base our approach on WSCoL, i.e., we associate basic events to specific locations of the BPEL specification and we allow them to access the values of the internal BPEL variables. A basic event is hence defined by a *declaration*, a *location* within the BPEL code and by a *property*. The declaration defines the name, event parameters, and type of the event (see below). The location consists of an XPath expression and of a keyword defining the kind of triggering condition we want (a pre- or a post-condition), as in the Dynamo approach. The property is defined in an extension of the WSCoL language which extends the types of values properties can evaluate to. More precisely, we identify the following three types of WSCoL expressions. *Boolean* expressions, i.e., WSCoL expressions describing boolean conditions; all the WSCoL expressions in the Dynamo approach, hence including the examples reported in Section 3, are of this type. *Numeric* expressions, i.e., WSCoL expressions that evaluate to a numeric value; an example is the following expression for event `ratio(uId: string): numeric`, which computes the ratio between the current and the previous insulin dose:

```
let $doseNew = ($labResults/suggestedDose);
let $doseOld = retrieve(pID, uID, iID,
  '[INVOKE]changeDose/post-condition', 'doseStored', 1);
let $ratio = $doseNew / $doseOld;
$ratio;
```

Tick expressions, i.e., WSCoL expression that express the fact that a given event has occurred; these expressions are useful if a given event, associated to a given BPEL location has to be reported to higher level monitors only under certain conditions; an example is the following expression for event `ratioOutOfBounds(uId: string): tick`, which reports an event only if the insuline ratio is out of bounds:

```
let $doseNew=...; $let $doseOld=... ; $let ratio=...;
($ratio>0.95 && $ratio<1.05 ? NOTICK : TICK)
```

Notice that the `NOTICK` keyword can be used also with numeric or boolean expressions, in case the valued event has to be reported only under certain condition; an example is the following expression for event `ratioIfOutOfBounds(uId: string): numeric`, which reports the insulin ratio only if it is out of bounds:

```
let $doseNew=...; $let $doseOld=... ; $let ratio=...;
($ratio>0.95 && $ratio<1.05 ? NOTICK : $ratio)
```

Composite monitor properties. While basic events are based on Dynamo, the combination of these basic events into complex monitoring properties is based on the ASTRO approach. That is, we replace the basic events described in Section 4 with the

events just introduced, while we keep the same operators and formulas defined in Section 4 for instance monitors and class monitors. More precisely, the syntax for events is "*name*`(%`*corr* `=` *par*`, ...)`", where *name* is the name of the event as defined in the WSCoL expression, *par* is the name of a parameter of the WSCoL declaration, and *corr* is a correlation variable, which is used to correlate events of different processes in a class monitor. Assume for instance that two basic events have been defined for service `TA`, namely `ratio (uId: string): numeric`, and `hospitalisation(uId: string): tick`. Then, the following *instance* monitor checks whether hospitalisation has been decided after the insulin dose has been incremented three times:

```
hospitalisation & count(Once(ratio > 1)) >= 3
```

The following *class* monitor reports the number of cases in which such an increment in the dose has lead to a hospitalisation for a given patient:

```
Count(hospitalisation(%pat = uId) &
        count(Once(ratio(%pat = uId) > 1)) >= 3)
```

Notice the usage of the correlation variable to select only the service executions corresponding to the same patient. If the correlation variable is removed, then a total count for all patients is computed:

```
Count(hospitalisation & count(Once(ratio > 1)) >= 3
```

We remark that the explicit correlation mechanism we adopt allows for the definition of cross-process class monitors. Indeed, it is easy to correlate events defined in different processes. Assume for instance that a different service monitors blood pression and defines basic events `lowPression/normPression(uId: string): TICK`. If we want to monitor the case where an increase in the insulin rate is recommended after a low blood pression is reported, we can define the following monitor:

```
(ratio(\%pat = uId) > 1) &
   ((!normPression(\%pat = uId)) Since (lowPression(\%pat = uId)))
```

Architecture. Since the monitoring language we have defined combines the localised basic events *à la* Dynamo with instance and class monitors approach *à la* ASTRO, the monitoring architecture is also a combination of Dynamo and ASTRO. More specifically, the computation of basic events is achieved by instrumenting the BPEL processes. This requires a slight modification of the code that is weaved by Dynamo into the execution engine. The instrumented processes, which are then deployed and executed on a standard BPEL engine, send information on the occurrence and value of basic events to the monitoring engine. This communication may occur through both asynchronous events and synchronous communications, allowing for both asynchronous and synchronous work by part of the execution and monitoring engine. The monitoring engine performs a correlation step in order to decide which monitor instances are relevant for a given basic event (and possibly instantiates new monitors if necessary). The monitoring engine also updates the status of the relevant monitor instances and reports problems and violations. The generation of the run-time components for the monitors combines the Dynamo and ASTRO approaches as well. Indeed, the former approach is exploited for instrumenting the BPEL processes, while the latter approach is responsible of generating the monitor instances executed by the monitoring engine.

7 Conclusions

In this paper, we proposed a novel monitor approaches that leverages Dynamo and ASTRO, two existing approaches for the run-time monitoring of BPEL orchestrations which have been developed within the research groups of the authors of this paper. The new approach is able to monitor a wider range of features than any other monitoring approach for BPEL orchestration the authors are aware of.

Our future work will concentrate on the implementation of the proposed integrated approach and on its thorough evaluation on real-world applications; this will also require detailing the definition of the novel language and of the architecture described in Section 6. On the side, we are also interested in evaluating the possibility to integrate other approaches that were not developed by the authors. In a longer term, we plan to investigate extensions of the proposed approach that include reactions to the anomalies and emergencies identified by the monitoring; the goal is to close the loop and influence the behaviour of the services according to the information collected during the monitoring. We also plan to extend the monitoring approach to the case of compositions of web services that are distributed among different nodes of the network; in this case, both the monitoring language and the monitoring architecture will have to deal with the necessity of collecting events over the network.

References

1. ActiveBPEL. The Open Source BPEL Engine, `http://www.activebpel.org`
2. Barbon, F., Traverso, P., Pistore, M., Trainotti, M.: Run-Time Monitoring of the Execution of Plans for Web Service Composition. In: Proc. ICAPS 2006, pp. 346–349 (2006)
3. Barbon, F., Traverso, P., Pistore, M., Trainotti, M.: Run-Time Monitoring of Instances and Classes of Web Service Compositions. In: Proc. ICWS 2006, pp. 63–71 (2006)
4. Baresi, L., Ghezzi, C., Guinea, S.: Smart monitors for composed services. In: Proc. ICSOC 2004, pp. 193–202 (2004)
5. Baresi, L., Guinea, S.: Towards dynamic monitoring of WS-BPEL processes. In: Benatallah, B., Casati, F., Traverso, P. (eds.) ICSOC 2005. LNCS, vol. 3826, pp. 269–282. Springer, Heidelberg (2005)
6. Baresi, L., Guinea, S.: A dynamic and reactive approach to the supervision of BPEL processes. In: Proc. ISEC 2008, pp. 39–48 (2008)
7. Beeri, C., Eyal, A., Milo, T., Pilberg, A.: Monitoring business processes with queries. In: Proc. VLDB 2007, pp. 603–614 (2007)
8. Bianculli, D., Ghezzi, C.: Monitoring Conversational Web Services. In: Proc. IW-SOSWE 2007 (2007)
9. Mahbub, K., Spanoudakis, G.: Run-time Monitoring of Requirements for Systems Composed of Web Services: Initial Implementation and Evaluation Experience. In: Proc. ICWS 2005, pp. 257–265 (2005)
10. Momm, C., Malec, R., Abeck, S.: Towards a Model-driven Development of Monitored Processes, Wirtschaftsinformatik, vol. 2 (2007)
11. Roth, H., Schiefer, J., Schatten, A.: Probing and monitoring of WSBPEL processes with web services. In: Proc. CEC-EEE 2006 (2006)