Business Protocol Monitoring

Samir Sebahi, Mohand-Said Hacid
Université de Lyon
Université Claude Bernard Lyon 1
LIRIS CNRS UMR 5205
France
{samir.sebahi | mohand-said.hacid}@liris.cnrs.fr

Abstract—**Because it is never sure that a business process successfully tested or statistically checked will have the expected behaviour during its execution, it is necessary to bring verification to the execution phase, by continuously observing and checking the correct behaviour of business processes during run-time. In this paper, we propose a new monitoring framework to monitor business protocols. We provide a monitoring language called BPath, which is an XPath-based language for both expressing and checking temporal and hybrid logical properties at run-time, making visibility on business process external behaviour by expressing and evaluating statistical queries over execution traces.**

Keywords-monitoring; business process; business protocol; XPath; hybrid logic

I. Introduction

The advent of web services and Service Oriented Architecture (SOA) has made a considerable progress in the way applications are developed and used, leading to the opening of new borders for information systems, with more automation of tasks, complex and multiple interconnection scenarios between applications within the same system and across different systems. In this context, the task of checking correctness of business processes at run-time becomes particularly challenging.

Currently, the common practice for developing service-based systems is to employ the SOA paradigm [1], which enables composition of services into business processes in a particular order and according to a set of rules to provide support for business processes.

Two features characterizing SOA have retained our attention and guided our investigation towards building an approach for monitoring business processes: SOA uses a message-based communication model, and most of specifications and languages used in SOA are XML based.

Based on these considerations, we designed and developed a new monitoring framework based on message abstraction. This abstraction is called business protocol [2]. We provide an extension of XPath [3] to accommodate verification issues. The resulting language (called BPath) is also a query language that can be used to track and make visibility on business process execution.

The paper is organized as follows: In Section II, we present some related works. Section III presents the concept of business protocol, and presents a monitoring scenario. Section IV describes architectural and design principles of our approach for monitoring. In Section V, we present our monitoring language. Then, we show its applicability to monitoring in Section VI. Finally, we conclude in Section VII by summarizing our work and identifying some extensions.

II. Related Work

A lot of research works have been proposed in the last years to monitor business processes. Some of them are directly related to our work. Baresi and Guinea [7] proposed a language (WSCoL) for specifying constraints on execution by defining a set of monitoring rules for both functional and non-functional constraints with the capability of setting the degree of monitoring at run-time such as: validity time frame, priority and a set of certified providers, for which monitoring may be omitted. Also, it enables specifying expressions over process variables and supports a set of built-in functions, logical and mathematical operators, and quantification. This work was extended in [8] by providing a support for specification and checking of temporal properties at run-time like with our monitoring framework. In [9], both business process behaviors and monitoring properties were stated as event calculus predicates, which is a logic-based formalism representing actions and their effects. Then, monitoring properties are checked in a post-mortem way against the stated behaviors and the recorded behavior in execution log at runtime, making the monitoring framework non intrusive regarding the execution of the business process, which is also the same case in our monitoring framework. The authors in [10][11] proposed monitoring languages that are built on top of XPath. [10] Proposed an approach to the monitoring of business processes specified in BPEL. A visual language, called Business Process Query Language (BPQL), with query capabilities, over BPEL processes, was introduced. XQuery expressions are generated, in the same way that graphical notations help business process designers generate specification code, using dedicated icons for each activity. Hallé and Villemaire [11] proposed an approach for monitoring web services choreography by means of XQuery [12] engine. Linear temporal logic properties are translated into an equivalent XQuery expression. Then, it is evaluated over XML message traces representing the choreography. Our monitoring framework is distinguished by using a simple messages based abstraction, and an expressive hybrid logic based language.

III. BUSINESS PROTOCOL

The purpose of a business protocol is essentially to specify the set of conversations (sequence of messages) that are supported by a business process [2]. Formally, we define a business protocol as a tuple $P = (S, s_0, F, M, T)$ where:

- S is a finite set of states the process goes through during its execution.
- s_0 is the initial state.
- F represents the finite set of final states.
- M is a set of messages.
- $T \subseteq S\times S\times M$ is the set of transitions, where every transition is labelled with a message name and its polarity, when a message is consumed by the protocol, the transition is assigned the polarity sign(+), and when it is produced by the protocol, the transition is assigned the sign (-).

In order to give an intuitive idea about our monitoring approach, let us consider the following scenario, inspired from [9], of an online Car Rental System (CRS) shown Figure 1.

CRS offers a car location service: whenever a rent car request is received *(RentCar)*, the availability of the requested car will be checked. If it is not available, then a list of cars will be sent to the client, otherwise, the requested car is reserved, and a confirmation message is sent to the client *(CarReservation)*. Then, the client will send her/his bank information (*BankInfo*), which will be validated, before sending the *keys*. After returning the keys, the client receives a payment confirmation *(BankConfirmation)*. But, in case the bank information is not valid, *CardRejected* message will be sent to the client and the process instance is completed.

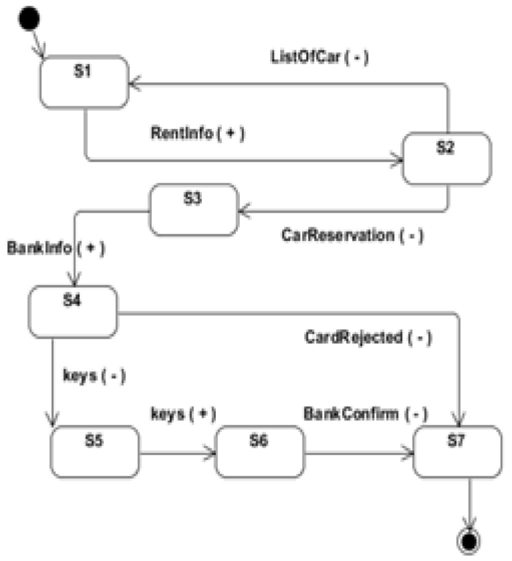

Figure 1. CRS business protocol

To show how our monitoring framework is able to monitor different kinds of properties and queries, we propose to consider the following list which should be continuously evaluated at run-time:

- **P1:** if a client's bank information is rejected, he should not get a car reservation before one hour.
- **P2: a** client should not get a car reservation when the keys are taken by another client.
- **Q1:** calculates the average time to perform a car reservation.
- **Q2:** counts the number of rejected bank information.

IV. THE OVERALL ARCHITECTURE

Figure 2 depicts the main components of the monitoring framework. First, a BPEL business process external behaviour is represented by means of a business protocol. Then, monitoring properties and queries are formulated using BPath monitoring language (presented in Section V).

At run-time, all incoming or outgoing messages will be captured by the business protocol monitor component before reaching their original destination. The process engine as well as the monitoring framework will publish the execution and monitoring events respectively, which will be stored in the execution log.

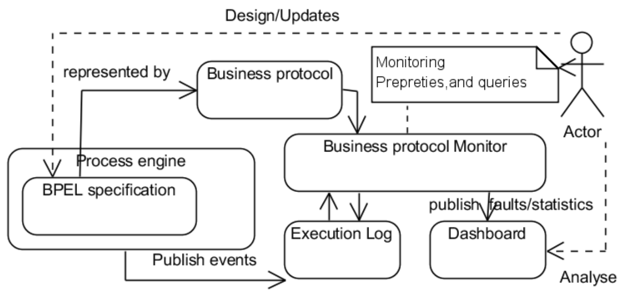

Figure 2. Monitoring framework

The execution log is of two types: state log, generated by the business protocol monitor, and event log generated by the process engine. On the basis of these generated execution logs, a checker component will check the correctness of the current execution and a Business activity monitor component will evaluate the specified statistical query to return statistical indicators on the execution of the process, and then both of these monitoring results will be published on a dashboard.

Additionally, the monitoring framework provides a set of business protocol execution events (see TABLE I.), to capture and control the exchanged messages, but also to specify when verification tasks should be performed.

For instance, to perform verification every time a message is received, we write:

OnMessageReceived (EventArgs e){
    Check a property(P_i)
}

Or after a message is sent, as follows:

OnMessageSent(EventArgs e){
    Check a property(P_j)
}

In the first case, a business process will be blocked until the verification is done. But, in the second case, verification task will not block the execution of the business process.

TABLE I. BUSINESS PROTOCOL EVENTS

Protocol Events	Description
OnEvent	Fires every time an event listed in this table occur.
OnNewInstance	Occurs when a new instance is started
OnNewState	Occurs when a state is entered
OnMessage	Occurs when a message sent or received
OnMessageReceived	Occurs when a message is received
OnAnknwonMessage	Occurs when a received message is not defined in the protocol
OnUnexpectedMessage	Occurs when a received message is defined in the protocol, but not expected from the current state
OnMessageSent	Occurs when a message is sent
OnTransition	Occurs when a transition from a state to another state happen
OnEndInstance	Occurs when an instance is ended

V. MONITORING LANGUAGE

In what follows, we consider that an execution of a business process as a sequence of states, independently of the fact that a business process can have different process instances, or parallel activities inside the same process instance.

The main idea behind our monitoring language (BPath) is first to consider a sequence of states representing the execution of a system as a special kind of tree. Each node represents a possible state, and its child node represents the direct next state. Then try to reuse the widely used language in the area of service based systems, which is XPath, as both a verification language and a query language.

So, BPath is built on top of XPath, and evaluated over a special tree of nodes (each node has only one child node, and no sibling nodes) forming a linear structure. BPath accommodates the notion of static and dynamic attributes and allows variable assignment inside path expressions. BPath offers a mean to express properties in first order hybrid logic. First order Hybrid logic [6] is an extension of first-order modal logic that makes it possible to name states and to assert that a formula is true at a named state.

A. BPath Syntax

A BPath formula is built according to the following abstract syntax:

$\varphi ::= T \mid T = T \mid P(T, \ldots, T) \mid \text{not } \varphi \mid \varphi \text{ and } \varphi \mid \varphi \text{ or } \varphi \mid (\varphi) \mid @_s\varphi \mid \downarrow s,\varphi \mid \downarrow_\pi s,\varphi \mid \downarrow_T x,\varphi \mid \exists x\ \varphi \mid \forall x\ \varphi$

$T ::= \pi \mid x \mid c \mid f(t_1,\ldots,t_n) \mid \pi/@q \mid s \mid s/@q$

$\pi ::= Axis::N \mid (\pi) \mid \pi\,[\varphi] \mid \pi / \pi \mid$

$N ::= n \mid *$

Axis ::= child | descendant | self | descendant-or-self| parent | ancestor| ancestor-or-self

Where: $x \in$ FVAR (a set of first-order variables), $n \in$LAB (a set of first-order constants), We define a function *lablel*: W→LAB, such that for each element of *W* associates an element of LAB, $s \in$ SVAR (a set of state variables), $q \in$ATTS (a set of unary function symbols, called static attributes) $\cup$ ATTD (a set of unary function symbols, called dynamic attributes) $\cup$ FUN (a set of one or more arity functions).

To simplify some expressions, we consider that *"π/child::N"* can be written as "π/N", that *"self::*/@q"* can be simply written as *"@q"*, and that "not (φ) ∨ α" can be abbreviated as "φ→ α".

B. BPath semantic

BPath formulas are interpreted in first-order modal models $M\ (W, R, D, I_w)_{w\in W}$ with constant domains such that: *W* is a set of nodes (or states)$\{w_1, w_2\ldots\}$, R is a linear modal relation on *W*. *D* is the interpretation domain. *(W, R)* is the modal frame.

For every $w \in W$, (D, I_w) is an ordinary first-order model such that:

- $I_w(c) = I_{w'}(c)$, for all w, w' $\in$ W, c$\in$ CON.
- $I_w(q) \in D$, for q ∈ ATTS ∪ ATTD ∪ FUN.
- $I_w(P) \subset D^k$, for P a k-ary predicate symbol.
- $I_w(\pi) \subset W$ for π a path expression.

To interpret formulas with free variables, we define an assignment function *g* such that:

g: SVAR × FVAR → W × D

$g(x) \in D$ if $x \in$FVAR or $g(x) \in W$ if $x \in$ SVAR.

Given a model and an assignment *g*, the interpretation of the term *t*, denoted by $\overline{t}$ is defined as:

- $\overline{x}$= g(x) for $x \in$ FVAR
- $\overline{s}$ = g(s) for $s \in$ SVAR
- $\overline{c} = I_w(c)$ for $c \in$ CON, for some $w \in$ W
- $\overline{s/@q} = I_{g(s)}(q)$ for *s* a state variable, and $q \in$ ATTS ∪ ATTD ∪ FUN.
- $\overline{\pi/@q} = \{I_{w1}(q)\ ,\ I_{w1}(q)\ ,\ldots\ I_{wn}(q)\ \}$such that $w_1,w_2\ldots w_n \in I_w(\pi)$, for some $w \in$W.

We also define an assignment $g^{x1..xn}_{d1\ldots dn}$ such that:

$$g^{x1..xn}_{d1\ldots dn}(y) = g(y) \text{ for } y \neq x_i, \text{ and } g^{xi..xn}_{d1\ldots dn}(x_i) = d_i.$$

This means that d_1 is assigned to x_1, d_2 to x_2…and for each *y* not in$\{x_1\ldots x_n\}$, $g^{x1..xn}_{d1\ldots dn}$ is the same as *g*.

The satisfaction relation of a BPath expression is defined as follows:

M, g, w |= t ⇔ $I_{M,w,g}(t) \neq \emptyset$.

M, g, w |= P($t_1, \ldots, t_n$) ⇔ $(\overline{t1}, \ldots, \overline{tn}) \subset I_w(P)$

M, g, w |= t = u ⇔ $\overline{t} = \overline{u}$, where: t and *u* are terms.

M, g, w |=not φ ⇔ M, g, w |≠ φ.

M, g, w |=φ and ψ ⇔ M, g, w |= φ ∧ M, g, w |= ψ.

M, g, w |= φ or ψ ⇔ M, g, w |= φ ∨ M, g, w |= ψ.

M, g, w|=$@_s\varphi$ ⇔ M, g, g(s) |= φ for s ∈SVAR.

M, g, w |=↓s,φ ⇔ M, g_{w}^{s} , w |=φ.

M, g, w |=$\downarrow_\pi$s,φ ⇔ M, $g_{w1,w1...wn}^{s,s[1]...s[n]}$, w |=φ. Where $w_{1...}w_n \in I_{M,g,w}(\pi)$.

M, g, w |=$\downarrow_T$x,φ ⇔ M, g_{T}^{x} , w |=φ.

M, g, w |=∃x φ ⇔ M, g_{w}^{x} , w |=φ. for some d ∈ D.

M, g, w |=∀x φ ⇔ M, g_{d}^{x} , w |=φ. for all d ∈ D.

The interpretation of a path on the model *M*, starting from the state-node *w*, and given an assignment *g* is defined as follows:

$I_{M,w,g}(\pi_1|\ \pi_2)= I_{M,w,g}(\pi_1) \cup I_{M,w,g}(\pi_2)$.
$I_{M,w,g}(\pi_1 \cap \pi_2)=\{w'|\ w' \in I_{M,w,g}(\pi_1) \wedge\ w' \in I_{M,w,g}(\pi_2)\ \}$.
$I_{M,w,g}(\pi_1/\pi_2)=\{w''|w' \in I_{M,w,g}(\pi_1)_w \wedge w'' \in I_{M,w',g}(\pi_2)\}$.
$I_{M,w,g}(\pi\ [\varphi])=\{w'|w' \in I_{M,w,g}(\pi) \wedge M,g,w'|= \varphi\ \}$.
$I_{M,w,g}$ (*self*::N) ={w| *label(w)*=N ∨ N=* }.
$I_{M,w,g}$ (*child*::N)={ (w'| (wRw'∧ ∀w'' wRw''→ w'Rw)) ∧ (*label(w')*=N ∨ N=*) }.
$I_{M,w,g}$(*descendant*::N) ={w'|wRw' ∧ (*label(w')*=N ∨ N=*) }.
$I_{M,w,g}$(*descendant-or-self*::N)= $I_{M,w,g}$(descendant::N) ∪ $I_{M,w,g}$ (*self*::N).
$I_{M,w,g}$ (*parent*::N)={ w'| (w'Rw∧ ∀w'' w''Rw→ w''Rw')) ∧ (*label(w')*=N ∨ N=*) }.
$I_{M,w,g}$ (*ancestor*::N)={w'|w'Rw ∧ (*label(w')*=N ∨ N=*) }.
$I_{M,w,g}$ (*ancestor-or-self*::N)= $I_{M,w,g}$ [ancestor::N] ∪ $I_{M,w,g}$ [self::N].

C. From BPath to XPath

To be evaluated, a BPath expression will be translated into a standard XPath expression, extended with two functions: *Set*, and *Get*, which allow to assign variables and to retrieve their values respectively.

The following table shows, the concrete BPath syntax, and how it is translated to XPath.

TABLE II. BPATH TO XPATH

BPath Abstract Syntax	BPath Concrete Syntax	Translation to XPath 1.0
x (a free variable)	$x	Get($x, self::*)
$@_s$φ	$s [φ]	Get(φ, $s)
$\downarrow_T$x,φ	$x:=T,φ	Set($x,t,self::*) and φ
↓s, φ	$s*, φ	Set($s, self::*,self::*) and φ
$\downarrow_\pi$s, φ	$s:= π, φ	Set($s, π ,self::*) and φ
s/@q	$s/@q	Get(q, $s)
π/@q (q∈ FUN ∪ ATTD)	π/@q	Get(q, π, self::*)

Quantified expression cannot be expressed in XPath 1.0 [3]. It is possible by using XPath 2.0 [4], as follows:

∃x φ: some $x in D satisfies φ
∀x φ: every $x in D satisfies φ

Listing 1 presents the *Get* and *Set* functions. We suppose that '*Eval()'* is a function provided by the framework to evaluate an XPath expression. *ɵ(q,w)* is a function returning the value of a dynamic attribute, *g* is an array storing variables and theirs values.

```
Set($x, t, w){g[x]=Eval(t,w), return true ;}
Get(φ , $s) :{  return Eval(φ, g[s])}
Get($x, w){return g[x];}
Get(q, $s) { return Get(q, g[s])  }
Get(q, π, w) { return sequence:{Get(q,w') / w'∈ Eval(π,w)}
Get(q, w) { if q ∈ ATTS ∪ FUN return Eval (q,w) else  if
(q∈ATTD) return ɵ(q,w) }
```

Listing 1 *Get* and *Set* functions

D. Linear Temporal Logic with BPath

Linear temporal Logic is a special type of modal logic: it provides a formal system for qualitatively describing and reasoning about how the truth values of assertions change over time [5]. LTL provides four future operators with the following meanings: X(φ): φ should be true on the next state, F(φ): means that φ should be true at least once in the future, G(φ): φ should be true every time in the future, φ U ψ: φ has to be true at least until ψ, which is true now or in the future. These operators can be represented in BPath as follows:

- X(φ): child::*[φ].
- F(φ): descendant-or-self::*[φ].
- G(φ): not(descendant-or-self::*[not(φ)]).
- φ U ψ: $x*, F($y*, $x[F($y=self ::* ∧ ψ) ∧ G (F ($y=self ::*) → φ)]).

VI. APPLICATION SCENARIO

In this section, we will show, through a concrete execution scenario, how BPath can be used to monitor a business process execution.

Let us assume that the car rental system manages three cars *(RedCar, GreenCar, BlueCar),* and receives requests from three clients *(John, Mark and Bob),* that we consider as web services interacting with the CRS business process: First, John sends a request for red car. His credit card will be rejected, but he tries again and gets the car reservation. Mark requests a green car, gets a reservation and keys, and then receives a payment confirmation after returning the keys. Bob requests the same car as Mark and obtains a reservation.

At run time, messages exchanged between different instances of the process and external partners will be captured and stored in the event log.

Definition 1: An event log is a collection of entries el = (name, (key=value), (key=value)...., InsId, T), where: *name* is the name of the event, (key=value) ...are list of items and their values contained within the event, *InsId* is an Instance identifier of the process instance concerned with the event, *and T* is a timestamp recording the time the event occured. Listing 2 shows an example of an event log, generated from the supposed execution scenario of the CRS business process.

```
L1 : RentInfo: ClientInfo=John; CarInfo=RedCar, InstId=1, T=1
L2: CarReservation: carReserved=yes, InstId=1, T=3
L3: CardRejected: cardInfo=798799979879, InstId=1, T=5
L4: RentInfo: ClientInfo=Mark; CarInfo=GreenCar, InstId=2, T=8
L5: CarReservation: carReserved=yes, InstId=2, T=10
L6: BankInfo: cardInfo=798799979879, InstId=2, T=12
L7 : RentInfo: ClientInfo=John; CarInfo=BlueCar, InstId=3, T=15
L8: CarReservation: carReserved=yes, InstId=3, T=17
L9: Keys: keysOut=KY123, InstId=2, T=19
L10: RentInfo: ClientInfo=Bob; CarInfo= GreenCar, InstId=4, T=22
L11: CarReservation: carReserved=yes, InstId=4, T=24
L12: Keys: keysIn=KY123, InstId=2, InstId=2, T=26
L13: BankConfirm: payeConfirmed =yes, BankTransation=Trans0001,
InstId=2, T=28
```

Listing 2 Event log

Additionally, the business protocol will generate events related to transition from a state to another state, when a message is received or sent to or by an instance of the process. These events are stored in the state log.

Definition 2: A state log (SL) is an XML tree of nodes (states-nodes): w_1, w_2, w_3...where w_2 is the unique child node of w_1, w_3 the unique child node of w_2, etc. Each state-node has a name $s_j \in LAB/\ s_j = label(w_i)$,, and two attributes, *InsId* (instance identifier)$\in$ATTS, and *T (*timestamp*)*$\in$ ATTS.

Listing 3 shows the states log generated from the supposed execution scenario of the CRS business process.

A BPath expression will be evaluated over the state log. But as we can see, state log does not contain a lot of information about the execution, because the real events are stored in the event log. Execution information can be retrieved and linked to a state-node through dynamic attributes.

```
<S1  InstId="1" T="0">
 <S2 InstId="1" T="2">
  <S3 InstId="1" T="4">
   <S4 InstId="1" T="6">
    <S1 InstId="2" T="7">
     <S2 InstId="2" T="9">
      <S3 InstId="2" T="11">
       <S4 InstId="2" T="13">
        <S1 InstId="3" T="14">
         <S2 InstId="3" T="16">
          <S3 InstId="3" T="18">
           <S5 InstId="2" T="20">
            <S1 InstId="4" T="21">
             <S2 InstId="4" T="23">
              <S3 InstId="4" T="25">
               <S6 InstId="2" T="27">
                <S7 InstId="2" T="29"/>
               </S6>
              </S3>
      {…}
</S1>
```

Listing 3 State log

In BPath, the value of a dynamic attribute at state-node *w* is defined by a function θ *(q, w)*, which extracts the last value of *q* from the event log, before that state node w occurs, as follows:

```
θ(q,w):
Begin
   Let q∈ el1 / el1 ∈ Event log ∧ el1.InstId=Eval(@InstId, w) ∧
   el1 .T<Eval(@T, w) ∧ ∀ el2 ∈ Event log: q∈ el2 ∧ el2.InstId=
   el1.InstId ∧ el2.T<Eval(@T, w))→ el2.T< el1.T;
   return q.value;
End
```

For instance, the following BPath expressions, when evaluated at T>4, will return:

- *S1/S2/@ClientInfo={John}.*
- *S1/S2/S3/@ClientInf={john}.*
- *S1/S2/S3/@ carReserved={yes}.*

Now, the monitoring properties and queries presented in Section III can be expressed using BPath as follows:

a) Check that in case where credit card of a client is rejected, the client should wait one hour to be able to get a car reservation. We formulate this property in BPath as follows (P1):

G(self::S7[$S7*, @CardRejected →
not(F(self::S3[@CleientInfo=$S7/@ClientInfo and (@T-
$S7/@T)<60]))]).

In this property, we check that every time in the future a credit card of a client is rejected (can be checked at state *S7*), the concerned client should not get a car reservation (we check a state *S3* following the previous *S7*), knowing that the elapsed time (between *S3* and *S7*) is less than one hour.

b) A client should not get a car reservation when the keys are taken by another client. This property can be expressed using BPath as follows (P2):

G (self::S5[$S5*, F(self::S3[$S3*, @CarInfo =
$S5/@CarInfo] → $S5[F(slef::S7[@CarInfo =
$S5/@CarInfo and = @T< $S3/@T])]]).

In this property we express that whenever keys of a car is sent (at state *S5*). Then, every time in the future where a reservation for the same car is done (at state *S3*), it should be the case that the keys of this car were returned before (if there exist a state *S7* after *S5* but before *S3*, where the keys of the car are returned)

As we can see from the previous execution log, the properties (P1, P2) are violated respectively at:

- L8 (see event log): when John obtains a car reservation, knowing that his credit card was rejected less than one hour ago (see L3).
- At line L11: the green car was reserved for Bob (at L11), but this car is still assigned to Mark (L9), and the keys of the car are returned by Mark only after (L11), exactly at (L12).

BPath is also a query language that can be used to return statistical indicators on the execution of a business process:

c) Calculating average time to make a car reservation (Q1):

sum(descendant-or-self::S1[$S1*, descendant::S3[$S3*, (@InstId=$S1/@InstId)]]/@($S3/@T-@T)) div count(descendant::S3).

In this query we start by calculating the sum for all process instances of the time to reach the state *S3* (the reservation state) from the state *S1* (the start state), then dividing the obtained sum on the number of reservations. We use two functions (sum and count) to respectively calculate the sum and the number of elements of a sequence.

d) Count the number of rejected credit cards we write in BPath (Q2):

Count(descendant-or-self::S7[@CardRejected]).

The previous list of monitoring properties and queries provides an overview on how to use BPath to monitor business processes. Additional functionalities can be expected when using BPath within XQuery, and by adding new built-in functions.

VII. Conclusion and Future Work

In this work, we provided a framework for business protocol monitoring. First, we have presented the business protocol abstraction. Then, we have presented BPath, the underlying monitoring language. Finally, through a case study, we have shown how the monitoring framework can be used to monitor business protocol. To summarize, we have developed a monitoring framework that mainly displays the followings features:

- The use of a simple messages based abstraction.
- A single expressive language for expressing both monitoring properties, and queries. However BPath is familiar to those who already use XPath language.
- Monitoring properties and queries can be dynamically specified during the execution of the process,
- Non-intrusive monitoring framework, because it is executed in completely separated way from the business process.

Our future work will be devoted to the design of methods to analyze and explain the reason of the deviations, and move towards resolving them as soon as they occur.

References

[1] A. Metzger and K. Pohl, "Towards the Next Generation of Service-Based Systems: The S-Cube Research Framework," Advanced Information Systems Engineering, 2009, pp. 11-16.

[2] B. Benatallah, F. Casati, and F. Toumani, "Analysis and Management of Web Service Protocols," Conceptual Modeling – ER 2004, 2004, pp. 524-541.

[3] J. Clark and S. DeRose, XML Path Language (XPath) Version 1.0, W3C, 1999.

[4] M. Kay, D. Chamberlin, J. Robie, M.F. Fernández, J. Siméon, S. Boag, and A. Berglund, XML Path Language (XPath) 2.0, W3C, 2007.

[5] E.A. Emerson, "Temporal and modal logic," Handbook of Theoretical Computer Science, 1995, pp. 995--1072.

[6] P. Blackburn and M. Marx, "Tableaux for Quantified Hybrid Logic," Automated Reasoning with Analytic Tableaux and Related Methods, 2002, pp. 259-286.

[7] L. Baresi and S. Guinea, "Towards Dynamic Monitoring of WS-BPEL Processes," Service-Oriented Computing - ICSOC 2005, 2005, pp. 269-282.

[8] L. Baresi, D. Bianculli, C. Ghezzi, S. Guinea, and P. Spoletini, "A Timed Extension of WSCoL," Web Services, IEEE International Conference on, Los Alamitos, CA, USA: IEEE Computer Society, 2007, pp. 663-670.

[9] K. Mahbub and G. Spanoudakis, "Run-time Monitoring of Requirements for Systems Composed of Web-Services: Initial Implementation and Evaluation Experience," IN ICWS '05, 2005, pp. 257--265.

[10] C. Beeri and A. Eyal, "Monitoring business processes with queries," IN VLDB, 2007.

[11] S. Hallé and R. Villemaire, "Runtime monitoring of web service choreographies using streaming XML," Proceedings of the 2009 ACM symposium on Applied Computing, Honolulu, Hawaii: ACM, 2009, pp. 2118-2125.

[12] D. Chamberlin, J. Snelson, J. Robie, and M. Dyck, XQuery 1.1: An XML Query Language, W3C, 2009.