Run-Time Monitoring of Instances and Classes of Web Service Compositions*

Fabio Barbon and Paolo Traverso
ITC-IRST
Via Sommarive 18, Povo
38050 Trento, Italy
[barbonfab,traverso]@itc.it

Marco Pistore and Michele Trainotti
DIT, University of Trento
Via Sommarive 14, Povo
38050 Trento, Italy
[pistore,trainotti]@dit.unitn.it

Abstract

The run-time monitoring of web service compositions has been widely acknowledged as a significant and challenging problem. In this paper, we propose a novel solution to the problem of monitoring web services implemented in BPEL. *We devise an architecture that clearly separates the business logic of a web service from its monitoring functionality. The architecture supports both "instance monitors" that deal with the execution of a single instance of* BPEL *process, as well as "class monitors" that report aggregated information about all the instances of a* BPEL *process. We also define a language for the specification of instance and class monitors. The language allows for specifying boolean, statistic, and time-related properties. Finally, we devise a technique for the automatic translation of all these kinds of monitors to Java programs.*

1. Introduction

The run-time monitoring of web service compositions has strong motivations. Indeed, even properties and requirements that are verified at design time, prior to deployment and execution, can be violated at run-time. This is especially the case of service oriented applications, which are most often developed by composing services that are made available by third parties, that are autonomously developed, and that can change without notification. Moreover, some problems can be detected only at run-time. There are indeed situations that, even if admissible at design time, must be promptly revealed when they happen, e.g., the fact that a bank refuses to transfer money to a partner on-line shop. This is also the case of all the statistical information that can be collected at run-time. For instance, the fact that the number of users that are not able to buy from an on-line shop suddenly increases can be the signal of unavailable products, or of problems with a bank that refuses the on-line payments. In both cases the occurrence of such a situation has to be reported as soon as possible to the business analyst, so that prompt reactions can be taken.

Several recent works have indeed started to address different aspects of the run-time monitoring of web services (compositions) and of distributed business processes, see, e.g., [12, 11, 3, 4, 8, 9]. In this paper we propose a novel solution to the problem of monitoring web service compositions and, in particular, to the monitoring of distributed business processes implemented in BPEL for web services [2]. The proposed solution has the following main features that differentiate it from the existing solutions.

We devise an architecture where the monitor engine and the BPEL execution engine are executed in parallel on the same application server. This allows for an integration of the two engines, still maintaining the two run-time environments distinct, and keeping the monitors clearly separated from the BPEL processes. As a result, differently from the framework proposed in [3, 4], we obtain a clear separation of the business logic from the monitoring task, which allows for an easier adaptation of the business process to the evolving business needs.

The architecture supports both instance and class monitors: *instance monitors* deal with the execution of a single instance of BPEL business process, while *class monitors* extract information from and/or check the behaviour of all the individual instances of a business process. For instance, an instance monitor can check if the bank has rejected the online payment during a specific session, while a class monitor can provide statistics about on-line payment rejections.

We provide a novel, rather expressive language for the specification of both instance and class monitors. The language allows for specifying boolean, statistic, and time-related properties to be monitored. Beyond monitors of

* This work is partially funded by the MIUR-FIRB project RBNE0195K5 "KLASE", "Knowledge Level Automated Software Engineering", by the MIUR-PRIN 2004 project "STRAP", and by the EU-IST project FP6-016004 "SENSORIA".

IEEE International Conference on Web Services (ICWS'06)
0-7695-2669-1/06 $20.00 © 2006 IEEE

IEEE COMPUTER SOCIETY

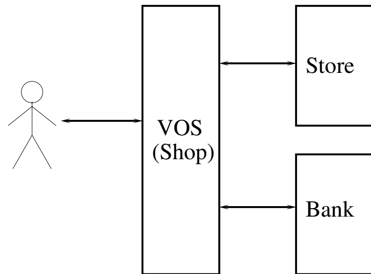


Figure 1. The Virtual Online Shop example

usual boolean properties, we can specify instance monitors that should, e.g., count the number of iterations that are executed in a given session, such as the number of times that a client changes the selected item to buy. We can specify that the monitor should issue an alert if the number of iterations exceeds a given threshold. Moreover, we can specify class monitors that collect information from all existing instance monitors, and check situations of interest — such as the fact that there has been at least one rejection of money transfer by the bank — and/or report on statistics — such as the percentage of payment rejections by the bank.

Finally, we devise a technique for the automatic generation of the code implementing the instance and class monitors, thus reducing the effort in their design and implementation. Monitors are automatically generated as Java programs that can be deployed in the run-time environment of the monitor engine. To the best of our knowledge, none of the exiting approaches for the run-time monitoring of web services support class monitors and their automated generation from high level specifications.

The paper is structured as follows. In Section 2, we introduce an explanatory example that will be used all along the paper. Section 3 describes the architecture of the run-time monitoring environment, while in Section 4 we describe the language for the specification of monitors, and how monitors are generated automatically from specifications in this language. In Section 5 we draw some conclusions and a comparison with related work.

2. An Example

In our explanatory example, the composed service is a Virtual Online Shop (VOS) that offers a combined sell and payment service to clients, by interacting with two external services: a Store and a Bank (see Figure 1). When the VOS receives a request for an item from a client, it contacts the Store, and, if the item is available, it gets back an offer including the price. In the case the client does not accept the offer, he/she can either terminate the interaction with the VOS, or require an offer for a different item. Once the client has accepted the offer, the VOS sends an acknowledgement to the Store, which in turn replies by sending back its bank account data. After obtaining also the client's bank account, the VOS invokes the Bank by sending the amount to be transferred from the client's account to the Store's account. The Bank performs the usual authentication procedures, and if the credential of the Store and of the client are not refused, the Bank sends back to the VOS a "transfer id" that identifies univocally the transaction and that the VOS routes to the Store. If the Store confirms the payment, the VOS confirms to the Bank, which performs the money transfer.

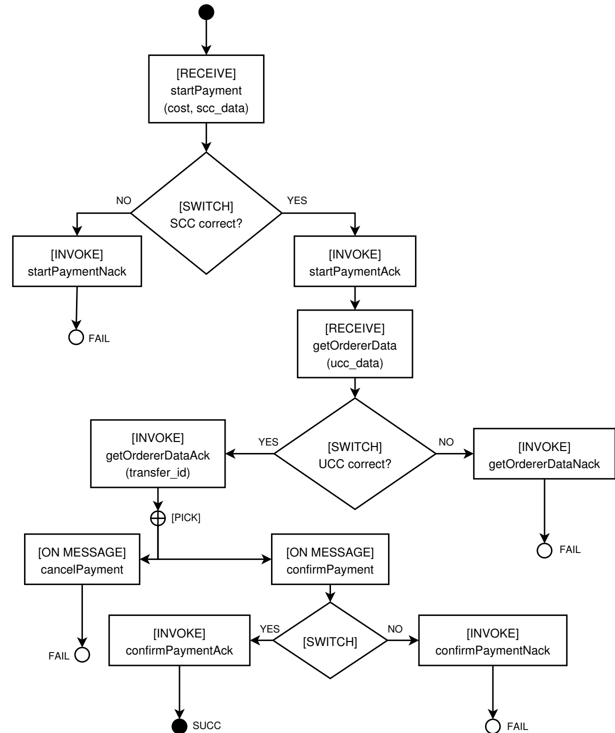


Figure 2. Bank abstract BPEL

In this example, the Store and the Bank are the external component services. We assume their abstract BPEL specifications, available on the Web, describe the interaction protocols that the VOS is expected to respect when interacting with them. In Figure 2 we show the interaction flow of the abstract BPEL of the Bank (the one for the Store is conceptually similar). This interaction is structured in three phases. In the first phase, the bank receives a request for a money transfer of a given amount (corresponding to the `cost` parameter of the message `[RECEIVE] startPayment`) to the bank account identified by the parameter `scc_data`. The second phase starts after the owner of the destination account, in our case the Store, has been validated (`[SWITCH] SCC Correct?`). In this phase, the Bank authenticates also the owner of the source account, in our case the VOS client (`[RECEIVE] getOrdererData`).

IEEE COMPUTER SOCIETY

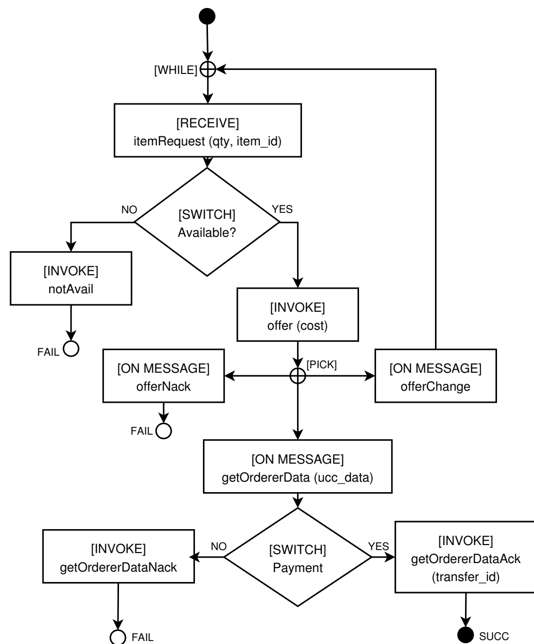


Figure 3. VOS abstract BPEL

The third phase starts when both accounts have been validated. The Bank sends back an acknowledgement and stops waiting for a confirmation or a cancellation. On confirmation (`[ON MESSAGE] confirmPayment`), the Bank can either refuse to perform the money transfer, for instance if there is not enough money on the source account (this is notified to the client with `[INVOKE] confirmPaymentNack`), or perform the transfer. In the latter case, a final acknowledgement is sent to the client of the Bank (`[INVOKE] confirmPaymentAck`).

The VOS is a new service that exploits the interfaces of Bank and Shop to provide the combined sell and payment service. Its abstract BPEL, depicted in Figure 3, defines the protocol interaction between the VOS and the client. The VOS becomes active upon a request for an item, which includes information about the quantity (see the `[RECEIVE] itemRequest` box). If the requested item is not available, the user is informed about this (`[INVOKE] notAvail` box) and the interaction terminates. Otherwise, the VOS sends an offer with a cost (`[INVOKE] offer` box). The availability condition (`[SWITCH] Available?`) depends on an interaction between the VOS and the Store that is hidden to the client. If the item is available, the VOS stops waiting for either a positive reply from the client (`[ON MESSAGE] getOrdererData`), or a negative response (`[ON MESSAGE] offerNack`). The client is given also the possibility to ask for another offer (`[ON MESSAGE] offerChange`). If the client accepts the offer, its message (`[ON MESSAGE] getOrdererData`) contains his/her bank account information for the payment. Finally, depending on whether the payment procedure is successful (this involves interactions of VOS with the Store and with the Bank), the client is notified with either an acknowledgement (`[INVOKE] getOrdererDataAck`) or a refusal (`[INVOKE] getOrdererDataNack`).

Despite the simplicity of the domain, there are several different properties that the provider implementing the Virtual Online Shop may want to monitor. A first class of properties are those that constrain the correct behaviors of the composition. An example is property:

- **StoreCcNotRefused**: the credentials of the Store are not refused by the Bank.

We remark that the interaction protocol with the bank allows for such a refusal; however, the VOS expects this refusal never to happen due to the contracts regulating the relations with the Bank. This refusal is an unexpected (and hence very dangerous) event that has to be detected at run-time as soon as it happens. This is an example of a boolean property that we want to check on all instances of the VOS process. Another boolean property is the following:

- **OfferBeforeBank**: the interaction with the Bank does not start before the User has accepted an offer.

The VOS may also be interested in counting how many times a given event occurs in the execution of a process instance. Examples of this properties are:

- **NotAvailCount**: count the number of times the Store reports that a requested item is unavailable.
- **RetriesOnSuccCount**: count the number of items offered to the User before the User accepts to buy.

Finally, the VOS may be interested in measuring the time spent to perform certain activities, for instance:

- **PaymentTime**: compute the time requested to finalize the payment with the bank.

All the properties described above are instance-level properties, that is, they are evaluated on the execution of one instance of the VOS service. On top of these properties, it is possible to define aggregated class-level properties, i.e., properties that consider all the instances of a business process.

- **GlobalStoreCcNotRefuse**: the credentials of the Store have never been refused by the Bank in any execution of the VOS.

This is an example of a boolean property for a class monitor. A related numerical property for a class monitor is the following:

- **CountStoreCCRefused**: count the total number of times the Bank has refused the credentials of the Store on all the executions of the VOS.

IEEE COMPUTER SOCIETY

The following two properties perform statistical analysis of properties related to the number of times a given event is repeated and of properties related to the time required to perform given activities:

- **AverageUserRetriesCount**: average number of times the user gets and refuses an offer from the VOS.
- **AveragePaymentTime**: average duration of the interactions with the back for the payment procedure.

3. The Run-Time Monitoring Environment

In our approach, monitors are software modules that run in parallel to BPEL processes, observe their behavior by intercepting the input/output messages that are received/sent by the processes, and signal misbehaviors or, more in general, situations or events of interest. In this section we describe how we implemented the run-time environment for the monitors within the specific BPEL platform that we adopted for our projects. The architecture that we describe, however, is modular and allows for a simple integration of our framework also in other BPEL platforms.

BPEL execution environment. We have chosen a standard engine for executing BPEL processes. Among the existing engines, we chose Active BPEL [1] for our experiments, since it is available as open source, and since it implements a modular architecture that is easy to extend.

From a high level point of view, the Active BPEL runtime environment can be seen as composed of four parts (see the light components of Figure 4). A *Process Inventory* contains all the BPEL processes deployed on the engine. A set of *Process Instances* consists of the instances of BPEL processes that are currently in execution. The BPEL *Engine* is the most complex part of the run-time environment, and consists of different modules (including the Process Manager, and the Queue Manager), which are responsible of the different aspects of the execution of the BPEL processes. The *Admin Console* provides web pages for checking and controlling the status of the engine and of the process instances.

For our monitoring purposes, the most relevant aspects in the execution of BPEL processes are the *creation* and the *termination* of a new process instance, and the *input* and *output* of messages. The engine manages to *create* a new instance for a BPEL process in the inventory when one of its start activities is triggered by an incoming message. The creation of the process instance is supervised by the Process Manager, and consists in the activation of the initial set of BPEL activities for that process. When all the activities of a process instance have been executed, the Process Manager *terminates* that instance. The Queue Manager is responsible for dispatching incoming and outgoing messages. When an *incoming message* is received, the engine tries to find an active process instance that matches the correlation data included in the message. If such an instance is found, then the message is stored in the "inbound queue" for that process instance, where it waits until it gets consumed by one of the activities of the process instance.[1] The management of *outgoing messages* is much simpler. The engine provides an "outbound queue", where outgoing messages are stored by invocation or reply activities. The Queue Manager is responsible for picking messages from the "outbound queue" and for dispatching them to the destination services.

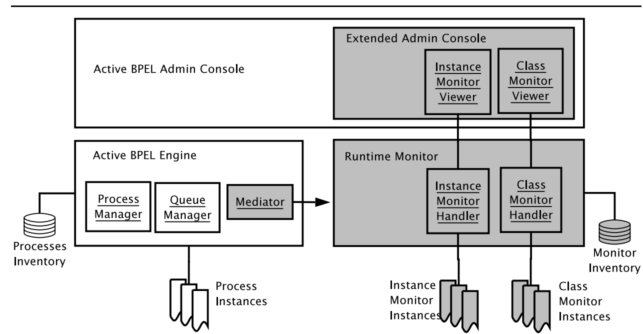


Figure 4. The Active BPEL engine extended with the run-time monitor environment

Run-Time Monitoring Environment. We have implemented the run-time monitoring environment as an extension of the Active BPEL environment.[2] In particular, we have extended Active BPEL with five new components (see dark part of Figure 4). The *Monitor Inventory* and the *Monitor Instances* are the counterparts of the corresponding components of the BPEL engine: the former contains all the monitors deployed in the engine, while the later is the set of instances of these monitors that are currently in execution. The *Run-Time Monitor (RTM)* is responsible to support the life-cycle (creation and termination) and the evolution of the monitor instances. The *Mediator* allows the RTM to interact with the Queue Manager and the Process Manager of the BPEL engine and to intercept input/output messages as well as other relevant events such as the creation and termination of process instances. The *Extended Admin Console* is an extension of the Active BPEL Admin Console that presents, along with other information on the BPEL processes, the information on the status of the corresponding monitors.

The framework supports two kinds of monitors. *Instance Monitors (IMs)*, which observe the execution of a single in-

1 If no matching instance is found, the message is parked in a "unmatched message queue" until a matching instance is found. For lack of space, we cannot discuss the details of the management of unmatched messages.

2 The implementation of the run-time monitoring environment is available from http://www.astroproject.org/ as open source code.

IEEE International Conference on Web Services (ICWS'06)
0-7695-2669-1/06 $20.00 © 2006 IEEE
IEEE COMPUTER SOCIETY

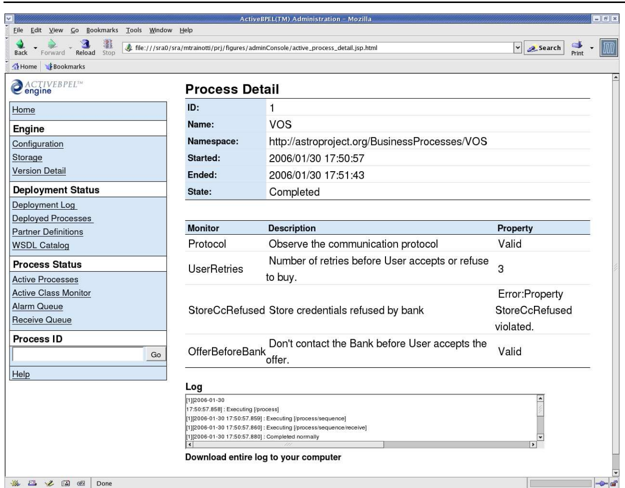


Figure 5. The IM Console

stance of a BPEL process; and *Class Monitors (CMs)*, which report aggregated information on all the instances of a given BPEL process. The two kinds of monitors are reflected in the architecture of the monitoring framework. Indeed, according to Figure 4, there are two distinct sets of monitor instances, namely *IM Instances* and *CM Instances*. In the RTM, the IMs and CMs are managed by two specific handlers, the *IM Handler* and the *CM Handler*. Also the Extended Admin Console provides two viewers, a *IM Viewer* and *CM Viewer*, to display the status of the two kinds of monitors. As an example, in Figure 5 we show one of the web pages generated by the Extended Admin Console, reporting the status of a specific instance of the VOS BPEL process. The console reports the status of the IMs associated to the BPEL process instance. This information, which is not present in the "standard" Active BPEL console, is generated by the IM Viewer.

Structure of a Monitor. A monitor is a Java class implementing the IMonitor interface described in Figure 6. More precisely, IMs implement interface IInstanceMonitor, while CMs implement interface IClassMonitor. The IMonitor interface defines four methods which are common to all monitors: `getProperty` and `getDescription` return a short and a long description of the property that is monitored; `getProcessName` returns the name of the BPEL process the monitors are associated to (the VOS in our exaple); and `getValue` returns the current value of the monitor (e.g., true or false in the case of a boolean monitor).

The methods defined by interfaces IInstanceMonitor and IClassMonitor manage the evolution of the monitors, and are better explained describing the life-cycle of instance and class monitors. The IMs life-cycle is influenced by three events: the process instance creation, the input/output

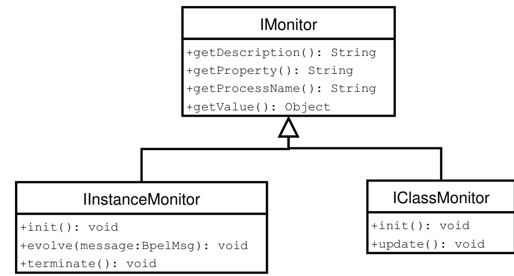


Figure 6. Methods of a monitor Java class

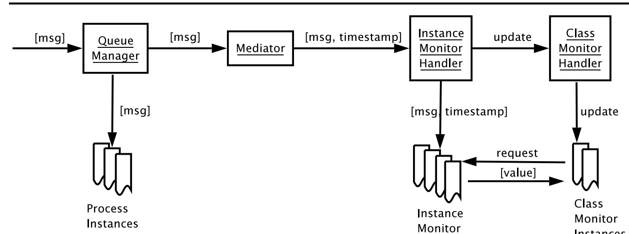


Figure 7. RTM message flow

of messages, and the termination of the process instance. When the RTM receives the notification of the creation of a new BPEL process, it creates a set of monitor instances that are specific for that process instance. The monitor instances are initialized through the method `init`. When the RTM receives a message from the Mediator, it sends it to the Instance Monitor Handler which dispatches the message to all the matching monitor instances through method `evolve`. For each message, the Mediator provides also information on the process instance receiving/sending the message, as well as on the BPEL process specification corresponding to the instance. The process termination is captured via a *termination event*, which is dispatched, through the invocation of method `terminate`, to all the monitor instances associated to the process instance.

The life-cycle of a class monitor is quite different. Method `init` is called only once, when the single instance of the class monitor is created. The evolution of the class monitor is triggered whenever the RTM receives a message or an event is received from any instance of the BPEL process to be monitored. More precisely, after the Instance Monitor Handler has dispatched the event or message to the relevant monitor instance, and this has been updated, it signals to the Class Monitor Handler that also the class monitors have to be updated. The Class Monitor Handler invokes method `update` on all the different class monitors associated to the BPEL process, which can update their internal status.

Figure 7 shows the flow on interactions triggered by the reception of a message by the BPEL engine. When the mes-

IEEE International Conference on Web Services (ICWS'06)
0-7695-2669-1/06 $20.00 © 2006 IEEE
IEEE COMPUTER SOCIETY

sage in received by the Queue Manager, a copy of it is forwarded to the Mediator. The Mediator marks the message with a time stamp and dispatches it to the RTM. The message is then passed to the Instance Monitor Handler which dispatches the message to the right IMs. The Instance Monitor Handler signals an update request to the Class Monitor Handler, which forces the update of the proper CMs. Figure 7 shows also how the CMs interact with the associated IMs in order to update their own states. Indeed, if a class monitor is responsible of collecting statistical data on all the instances of a given BPEL process, then it has to interact with the instance monitors that collect the non-aggregated values for the single process instances. This is achieved through the `request`/`[value]` flows shown in figure.

4. Automatic Generation of Monitors

In this section we introduce RTML, the Run-Time Monitor specification Language. As we will see, RTML is rather expressive: it allows for the specification of IMs as well as CMs; moreover, it allows for specifying boolean properties related to the execution of processes, as well as statistic properties and time-related properties. We also illustrate the technique we have devised for translating automatically RTML monitor specifications into the Java code that implements the monitors.

We provide the description of RTML in three steps. First of all, we define the language for specifying the "events" that are relevant for the evolution of monitors. On top of events, we define the language for specifying instance monitors. Finally, we define the language for class monitors on top of instance monitors.

Events. According to the framework of Section 3, the relevant events for monitors are:

- The creation and termination of a process instance; these two events are modeled through keywords "start" and "end" in RTML.
- The input and output of messages; in this case, RTML requires to specify the link on which the message is received or sent,[3] the fact that the message is an input or an output, and the message type. For instance "msg(VOS.output = offer)" corresponds to the shop sending an offer to the user, while "msg(Bank.input = startPayment)" corresponds to the bank receiving from the shop a request for starting a payment. It is also possible to specify constraints on the exchanged message values, as in "msg(Shop.input = request [item = Book])".

In some cases, it is preferable to speak of the effects of an event on the status of an interaction protocol, rather than of the event itself. For instance, let us consider the events leading to a termination of the interactions with the bank in a failure state: they correspond to msg(Bank.output = startPaymentNack), msg(Bank.output = getOrderDataNack), msg(Bank.output = confirmPaymentNack), and msg(Bank.input = cancelPayment). Alternatively, all these events can be described by "cause(Bank.state = FAIL)", that is, all the events that cause the bank to reach an activity named "FAIL". Similarly, we can say "cause(*link*.*var* = *val*)" to denote all events that cause variable *var* of BPEL process *link* to assume value *val*.[4]

The complete grammar for events e is the following:

$$\begin{aligned} e ::= &\ \mathsf{start} \mid \mathsf{end} \mid \\ &\ \mathsf{msg}(\mathit{link}.\mathsf{input/output} = \mathit{msg}[\mathit{opt\text{-}constraints}]) \mid \\ &\ \mathsf{cause}(\mathit{link}.\mathit{var} = \mathit{val}) \mid \\ &\ \mathsf{cause}(\mathit{link}.\mathsf{state} = \mathit{label}) \end{aligned}$$

Instance monitor formulas. The following grammar defines the formulas that specify instance monitors. We distinguish *boolean* formulas b, which monitor properties that can be either true or false, and *numeric* formulas n, which monitor properties that define a numerical value.

$$\begin{aligned} b ::= &\ e \mid \mathsf{Y}\,b \mid \mathsf{O}\,b \mid \mathsf{H}\,b \mid b\,\mathsf{S}\,b \mid \\ &\ n = n \mid n > n \mid \neg b \mid b \wedge b \mid \mathsf{true} \\ n ::= &\ \mathsf{count}\,(b) \mid \mathsf{time}\,(b) \mid b?n{:}n \mid \\ &\ n + n \mid n - n \mid n * n \mid n/n \mid 0 \mid 1 \mid \ldots \end{aligned}$$

A boolean formula can be an event e, a Past LTL [6] formula (operators Y, O, H, and S), a comparison between numeric formulas, or a logic combination of other boolean formulas. A numeric formula can be either a counting formula (operators count and time), a conditional expression ($b?n{:}n$), or an arithmetic operation on other numeric formulas.

Formulas are evaluated whenever a relevant event is received by the instance monitor. Formula e is true if it is compatible with the occurring event. Past LTL formulas have the following meaning:

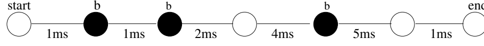
- $\mathsf{Y}\,b$ means "b was true in the previous step";
- $\mathsf{O}\,b$ means "b was true (at least) once in the past";
- $\mathsf{H}\,b$ means "b was true always in the past";
- $b_1 \mathsf{S}\, b_2$ means "b_1 has been true since b_2".

3 In the paper, we use as name of the link the name of the abstract BPEL process specifying the interaction protocol. That is, in the case of the VOS example, "Bank" refers to the messages between bank and shop, "Store" to the messages between store and shop, and "VOS" to the messages between shop and client.

4 We want to stress once more that we do not monitor the fact that the status of a process changes, or that a variable gets a value. We monitor the occurrence of an event that triggers such an effect.

IEEE COMPUTER SOCIETY

The meaning of the other boolean formulas is standard. Numeric formula $\mathsf{count}\,(b)$ counts the number of times that boolean condition b has been true since the creation of the process instance. Formula $\mathsf{time}\,(b)$ is similar, but it counts the sum of the time-spans after occurrences of condition b. Consider for instance the sequence of events modeled in the following diagram, corresponding to the execution of a fictitious BPEL process:



We have marked in black the events satisfying condition b, and we have annotated the duration of the time intervals between events. At the end of the execution we have $\mathsf{count}\,(b) = 3$ (b has been true 3 times) and $\mathsf{time}\,(b) = 8ms$ (corresponding to the sum of durations $1ms+2ms+5ms$).

The IM properties we have introduced in Section 2 can be defined by RTML following formulas:

- **OfferBeforeBank**:
  $\mathsf{msg}(\mathsf{Bank.input} = \mathsf{startPayment}) \Rightarrow$
  $\mathsf{O}\,\mathsf{msg}(\mathsf{Store.input} = \mathsf{offerAck}))$
- **NotAvailCount**:
  $\mathsf{count}\,(\mathsf{msg}(\mathsf{Store.output} = \mathsf{notAvail}))$
- **RetriesOnSuccCount**:
  $\mathsf{O}\,(\mathsf{cause}(\mathsf{VOS.state} = \mathsf{SUCC}))?$
  $\mathsf{count}\,(\mathsf{msg}(\mathsf{VOS.ouput} = \mathsf{offer}))) : 0$
- **PaymentTime**:
  $\mathsf{time}\,((\neg(\mathsf{cause}(\mathsf{Bank.state} = \mathsf{SUCC}, \mathsf{FAIL}))\mathsf{S}$
  $\mathsf{msg}(\mathsf{Bank.input} = \mathsf{startPayment})))$

We have implemented a procedure that translates automatically an instance RTML formula into the Java code implementing the monitor. The skeleton of the class is described in Figure 8. The key element in the generation of the Java code implementing the property is map `Val`, which associates a value to all the sub-formulas of the property to be monitored. Function `Init_RTML` assigns initial (truth or numerical) values to these sub-formulas, while function `Update_RTML` updates these values according to a received event. The update is done compositionally on the structure of the formula, and exploits the old values of the sub-formulas that are stored in variable `Val_old`. Function `Update_RTML` assigns values to Val as follows:

- $Val(e) :=$ "if event e is occurring"
- $Val(\mathsf{Y}\,b) := Val_{old}(b)$
- $Val(\mathsf{O}\,b) := Val_{old}(\mathsf{O}\,b) \vee Val(b)$
- $Val(b_1 \mathsf{S}\, b_2) := Val(b_2) \vee (Val_{old}(b_1 \mathsf{S}\, b_2) \wedge Val(b_1))$
- $Val(\mathsf{H}\,b) := Val_{old}(\mathsf{H}\,b) \wedge Val(b)$
- $Val(n_1 = n_2) := (Val(n_1) = Val(n_2))$
- $Val(n_1 > n_2) := (Val(n_1) > Val(n_2))$
- $Val(b_1 \wedge b_2) := (Val(b_1) \wedge Val(b_2))$
- $Val(\neg b) := \neg Val(b)$

```
package monitor.instance;
import org.astroproject.monitor.core.*;
import java.util.Map;
// Instance monitor for formula b
public class InstanceMonitor implements IInstanceMonitor {
  private Map Val; // Associates a truth value to each sub−formula

  public void init() {
    Init_RTML(Val); // Initialize Val according to event "start"
  }

  public void evolve(BpelMsg message) {
    Map Val_old = Val;
    Update_RTML(Val, Val_old, message);
    // Update Val compositionally on the sub−formulas
    // according to event "message"
  }

  public void terminate() {
    Map Val_old = Val;
    Update_RTML(Val, Val_old, EVENT_TERMINATE);
    // Update Val compositionally on the sub−formulas
    // according to event "end"
  }

  public Object getValue() {
    return Val.get("b");
  }

  public String getProcessName() { ... }
  public String getProperty() { ... }
  public String getDescription() { ... }
}
```

Figure 8. Skeleton for IM class

- $Val(\mathsf{count}\,(b)) := \text{if } Val(b) \text{ then } (Val_{old}(\mathsf{count}\,(b)) + 1) \text{ else } Val_{old}(\mathsf{count}\,(b))$
- $Val(\mathsf{time}\,(b)) := \text{if } Val(b) \wedge Val_{old}(b) \text{ then } (Val_{old}(\mathsf{time}\,(b)) + \mathsf{elapsed}) \text{ else } Val_{old}(\mathsf{time}\,(b))$
- $Val(b?n_1{:}n_2) := \text{if } Val(b) \text{ then } Val(n_1) \text{ else } Val(n_2)$
- $Val(n_1 + n_2) := (Val(n_1) + Val(n_2))$
- ...

In the case of operator time, value $\mathsf{elapsed}$ is the elapsed time from last event relevant for the monitor. This value is computed using the time-stamps associated to the events sent to the monitors.

Class monitor formulas. Also for the class monitors we distinguish among *boolean* formulas B and *numeric* formulas N, as shown by the following grammar.

$$B ::= \mathsf{And}\,(b) \mid \mathsf{Y}\,B \mid \mathsf{O}\,B \mid \mathsf{H}\,B \mid B\,\mathsf{S}\,B \mid$$
$$N = N \mid N > N \mid \neg B \mid B \wedge B \mid \mathsf{true}$$
$$N ::= \mathsf{Count}\,(b) \mid \mathsf{Sum}\,(n) \mid$$
$$N + N \mid N - N \mid N * N \mid N/N \mid 0 \mid 1 \mid ...$$

where b and n are instance monitor formulas.

Most of the operators are identical to those of the instance monitor formulas. We now describe the meaning of the operators specific of the class monitor formulas. Boolean formula $\mathsf{And}\,(b)$ checks if property b is true for all the instances of the BPEL process corresponding to the monitor. Numeric formula $\mathsf{Count}\,(b)$, instead, counts the number of instances of the BPEL process for which formula b holds. Numeric formula $\mathsf{Sum}\,(n)$ is similar, but aggregates numeric instance module formulas: it sums up the values of numeric formula n on all the instances of the BPEL process.

The CM properties we have introduced in Section 2 can be defined by RTML following formulas, where we used the abbreviation $\mathsf{Avrg}\,(n) = \mathsf{Sum}\,(n)/\mathsf{Count}\,(\mathsf{O}\,\mathsf{start})$ for computing the average value of numeric formula n:

- **GlobalStoreCcNotRefuse**:
  $\mathsf{And}\,(\neg\mathsf{O}\,\mathsf{msg}(\mathsf{Store.input} = \mathsf{startPaymentNack}))$
- **CountStoreCCRefused**:
  $\mathsf{Count}\,(\mathsf{O}\,\mathsf{msg}(\mathsf{Store.input} = \mathsf{startPaymentNack}))$
- **AverageUserRetriesCount**:
  $\mathsf{Avrg}\,(\mathsf{count}\,(\mathsf{msg}(\mathsf{VOS.output} = \mathsf{offer})))$
- **AveragePaymentTime**:
  $\mathsf{Avrg}\,(\mathsf{time}\,((\neg(\mathsf{cause}(\mathsf{Bank.state} = \mathsf{SUCC}, \mathsf{FAIL}))$
  $\mathsf{S}\,\mathsf{msg}(\mathsf{Bank.input} = \mathsf{startPayment}))))$

Also in the case of class monitors, we have implemented a translator from RTML to Java code. The key difference with respect to instance monitors is in the implementation of operators And, Count, and Sum. Indeed, these operators serve as link between class-level monitoring and instance-level monitoring. The translation algorithm adopts the following approach:

- An instance monitor is generated for each instance property b or n that appears as argument of operators And, Count, and Sum in the class formula.
- One class monitor is generated that aggregates the data of the instance monitors according to the formula.

The class monitor has the same structure as the instance monitor. In particular, it exploits map `Val` to associate a (boolean or numeric) value to each class-level sub-formula of the property to be monitored. Functions `init` and `update` are defined similarly to `init` and `evolve` in the instance monitor. (Function `terminate` is not present in the class monitors, since their execution is not supposed to terminate.) The main difference is in the implementation of function `Update_RTML`. We report its definition for operators And, Count, Sum:

- $Val(\mathsf{And}\,(b)) := \bigwedge_{i \in \mathcal{I}(b)} i.Val(b)$
- $Val(\mathsf{Count}\,(b)) := \sum_{i \in \mathcal{I}(b)} (\text{if } i.Val(b) \text{ then } 1 \text{ else } 0)$
- $Val(\mathsf{Sum}\,(n)) := \sum_{i \in \mathcal{I}(n)} i.Val(n)$

where $\mathcal{I}(b)$ is the set of all instances of the instance monitor corresponding to formula b, and $i.Val(b)$ is the value of formula b in monitor instance i.

In the implementation of $\mathsf{Count}\,(b)$, we have taken into account that $\mathcal{I}(b)$ contains monitors for *all* instances of a given BPEL process. This includes all BPEL instance that are already terminated, plus all running BPEL instances. As a consequence, $Val(\mathsf{Count}\,(b))$ has a *consolidated* component that takes into account the terminated BPEL instances, and an evolving component for the processes that are still running. The consolidate component is stored in an additional variable $Solid(\mathsf{Count}\,(b)$ and, whenever an update is performed, the value of $Solid(\mathsf{Count}\,(b))$ is updated taking into account the instances which terminated since the last update. Then the value of $Val(\mathsf{Count}\,(b))$ is computed adding to $Solid(\mathsf{Count}\,(b))$ the contribution of the instances that are still running. Similar considerations also hold for operators And and Sum.

5. Conclusions and Related Work

In this paper, we have presented a novel approach to the problem of monitoring web services described as BPEL processes. The approach allows for a clear separation of the service business logic from the monitoring functionality. Moreover, it provides the ability to monitor both the behaviours of single instances of BPEL processes, as well as behaviours of a class of instances. The monitors can check temporal, boolean, time related, and statistic properties. We devise a language that is expressive enough to express formally specifications of all these kinds of monitors, and a technique to automatically generate both instance and class monitors from their specifications, thus supporting their development.

Run-time monitoring has been extensively studied in different areas of computer science, such as distributed systems, requirement engineering, programming languages, and aspect oriented development, see, e.g., [5, 7, 10, 13]. There have been several proposals that deal with different aspects of the monitoring of web services and distributed business processes, see, e.g., [12, 11, 3, 4, 8, 9].

The works closest to ours are those described in [3, 4], and in [8, 9]. In [3, 4], monitors are specified as assertions that annotate the BPEL code. Annotated BPEL processes are then automatically translated to "monitored processes", i.e., BPEL processes that intearleave the business processes with the monitor functionalities. This approach allows for monitoring time-outs, runtime errors, as well as functional properties. On the one hand, an advantage of this approach is that monitors are themselves services implemented in BPEL, and can run on standard BPEL engines. Moreover, since the monitoring occurs within the BPEL process, if a monitored property fails, the monitor can influence the behavior of the BPEL process, e.g., forcing a termination. On the other hand, we allow for both instance and class monitors, as well as for the monitoring of properties and for the collection of information that depend on the whole history of the execution path. These kinds of monitors would be hard to express as assertions. Moreover, we allow for a clearer separation of the business logic from the monitoring task than in [3, 4], since we generate an executable monitor that is fully distinguished from the executable BPEL that runs the business

IEEE COMPUTER SOCIETY

logic. Finally, in general, our monitors can capture possible misbehaviors generated by the internal mechanisms of the BPEL execution engine.

The work described in [8, 9] shares with us the idea to have a monitor that is clearly separated from the BPEL processes. It provides a framework for monitoring behavioral properties and assumptions expressed in the event calculus. In addition to a different technical setting, the most important difference is that we allow for statistical and time-related properties and for class monitors, which are not supported in their setting.

In the future, we plan to extend the framework with techniques that allow for the automated failure-handling, repairing and adaptation triggered by information provided by monitors. Moreover, we plan to provide an experimental evaluation of the usability and of the practical effectiveness of the proposed technique on a set of application domains.

References

[1] ActiveBPEL. The Open Source BPEL Engine - http://www.activebpel.org.

[2] T. Andrews, F. Curbera, H. Dolakia, J. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weeravarana. Business Process Execution Language for Web Services (version 1.1), 2003.

[3] L. Baresi, C. Ghezzi, and S. Guinea. Smart Monitors for Composed Services. In *Int. Conf. on Service-Oriented Computing*, 2004.

[4] L. Baresi and S. Guinea. Towards dynamic monitoring of WS-BPEL Processes. In *Int. Conf. on Service-Oriented Computing*, 2005.

[5] A. Dingwall-Smith and A. Finkelstein. From Requirements to Monitors by way of Aspects. In *Int. Conf. on Aspect-Oriented Software Development*, 2002.

[6] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*. Elsevier, 1990.

[7] M. Feather and S. Fickas. Requirements Monitoring in Dynamic Environment. In *Int. Conf. on Requirements Engineering*, 1995.

[8] K. Mahbub and G. Spanoudakis. A Framework for Requirements Monitoring of Service Based Systems. In *Int. Conf. on Service-Oriented Computing*, 2004.

[9] K. Mahbub and G. Spanoudakis. Run-Time Monitoring of Requirements for Systems Composed of Web-Services: Initial Implementation and Evaluation Experience. In *Int. Conf. on Web Services*, 2005.

[10] D. Peters. Deriving Real-Time Monitors for System Requirements Documentation. In *Int. Symp. on Requirements Engineering - Doctoral Symposium*, 1997.

[11] W. Robinson. Monitoring Web Service Requirements. In *Int. Conference on Requirement Engineering*, 2003.

[12] A. Sahai, V. Machiraju, A. van Morsel, and F. Casati. Automated SLA Monitoring for Web Services. In *Int. Workshop on Distributed Systems: Operations and Management*, 2002.

[13] K. Sen, A. Vardhan, G. Agha, and G. Rosu. Efficient Decentralized Monitoring of Safety in Distributed Systems. In *Int. Conf. on Software Engineering*, 2004.

0-7695-2669-1/06 $20.00 © 2006 IEEE