INTERFACE EXPRESSIONS MONITORING FOR BPEL PROCESSES

Wassim Jendoubi, Guy Tremblay, Aziz Salah
Dépt. d'informatique, UQAM, C.P. 8888, Succ. Centre-Ville, Montréal, QC, H3C 3P8, Canada
jendoubi.wassim@courrier.uqam.ca, {tremblay.guy,salah.aziz}@uqam.ca

Keywords: BPEL Processes; Run-time Monitoring; Interface Protocols; Regular Expressions.

Abstract: In this paper, we show how Web services descriptions can be extended with simple *declarative* behavior specification using *interface expressions*, a form of regular expressions that describe the possible sequences of externally observable events that a WS-BPEL process can perform.
We describe how a concrete (executable) WS-BPEL process can be monitored with respect to such interface expressions, to ensure that it satisfies its associated abstract specification or to detect the occurrence, or non-occurrence, of some particular sequences of events. More specifically, we describe the implementation of such a run-time monitor, called BPEL.RPM, that uses the Open ESB BPEL service engine.

1 INTRODUCTION

In this paper, we show how BPEL Web services descriptions can be extended with simple declarative behavior specifications using a notation, similar to regular expressions, called "interface expressions." We also describe an architecture for monitoring, at run-time, a BPEL service implementation with respect to such interface expressions. Such a monitor can detect the occurrence of particular sequences of operations with respect to an interface expression, whether this expression specifies a "complete" contract, a particular sub-behavior, or a behavior that should not be observed.

2 WEB SERVICES INTERFACES AND PROTOCOLS

WS-BPEL (Andrews et al., 2003) is a notation for describing business processes implemented as orchestration of Web services.

A WS-BPEL description consists of two key parts: *static interface* and *dynamic behavior*. The static interface describes the various types of messages exchanged by the service as well as the collection of operations *provided* or *required* by the service. The dynamic behavior is described operationally, using typical (imperative) control structures.

`^`	Start of process
`$`	End of process
`any`	Arbitrary message (wildcard)
`in:m`	Reception of `m`
`out:m`	Sending of `m`
`outIn:m`	Synchronous invocation of `m`
`e1; e2`	Execution of `e1` followed by `e2`
`e1 [] e2`	Choice between `e1` and `e2`
`e1 || e2`	Arbitrary interleaving of `e1` and `e2`
`e?`	Optional element (0 or 1 occurrence of `e`)
`e*`	Repetition (0, 1 or many occurrences of `e`)
`e+`	Repetition (1 or many occurrences of `e`)

Figure 1: Notation for interface expressions.

The static interface describes a service as a collection of operations that can be invoked *in any particular order*, much like a Java `interface`. Thus, it does not describe the allowable *conversations*, i.e., the sequences of messages that can be ou should be exchanged between a client and a service. Such a *dynamic interface* specification is called a *protocol*.

Various notations can be used to describe protocols (Clements and al., 2003), e.g., UML's sequence, communication or state transition diagrams, regu-

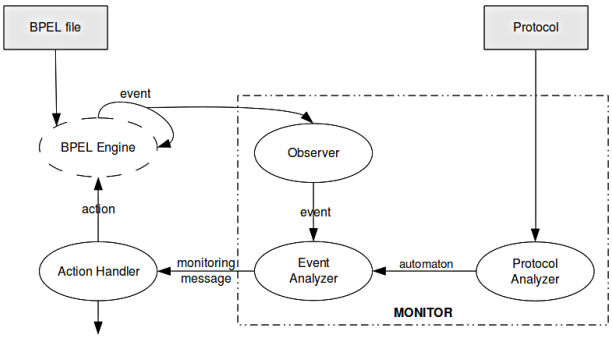

Figure 2: Overall architecture of our BPEL run-time monitoring system.

lar expressions, abstract WS-BPEL processes, etc. Figure 1 shows the key elements of one such notation, called *interface expressions* (Tremblay and Chae, 2005).

3 AN ARCHITECTURE FOR MONITORING WEB SERVICES BEHAVIOR

Run-time software monitoring can be used for various purposes, e.g., profiling, performance analysis, fault-detection, etc. (Delgado et al., 2004). In this paper, we propose a run-time monitor for behavior detection that can also be used for software-fault detection.

Figure 2 shows the overall architecture of our BPEL run-time monitoring system, expressed in a style similar to the one used by Delgado et al. Our architecture allows for the monitoring of various protocols expressed as interface expressions. The monitor notifies, dynamically, when a specific behavior has been detected—or when it cannot be detected.

The architecture we propose is independent of the specific BPEL engine used for running the BPEL processes. This architecture is also customizable and extensible with respect to how the monitor reacts when a behavior is detected.

We allow the monitoring of a process against various interface expressions, possibly all independently and simultaneously. Thus, given a process and a specific interface expression, there will be a specific monitor instance whose role will be to monitor the expected behavior.

4 MONITORING WS-BPEL PROCESSES USING BPEL.RPM

Figure 3 shows the overall structure of BPEL.RPM (*BPEL Run-time Process Monitoring*), a concrete implementation of the run-time monitoring architecture described earlier.

BPEL.RPM is composed of three key components:

- An *Event Observer*, which is specific to a particular execution engine.
- An *Analyzer*, which analyzes the various events as they occur and match them with respect to a specified interface expression.
- An *Action Handler*, which reacts according to some user-defined policy when a specific behavior is detected.

4.1 Interface Expression Specification

We showed earlier (Figure 1) an abstract notation for interface expressions, used to specify behavior protocols. In BPEL.RPM, we use a mixed regular expressions and XML representation, where a `Protocol` is characterized by two key elements:

- `MonitoredActivities`: The set of activities from the BPEL process which are monitored. Thus, an event associated with a BPEL activity not in this set is *ignored*.
- `InterfaceExpr`: The interface expression representing the behavior to be monitored. The outcome (verdict) of monitoring depends on the value of the `Excludes` attribute:[1]
  - `Excludes="no"`: Monitoring succeeds when an instance of the behavior specified by the `InterfaceExpr` is detected, and fails when it definitively cannot satisfy it. This case is typically associated with an explicit contract specification, but can also be used for identifying *interesting* sub-sequence of events, e.g., as required for intrusion detection.
  - `Excludes="yes"`: Monitoring succeeds when the observed behavior *does not match* the interface expression, and fails when it matches.

Monitoring using interface expression is *similar*, although not identical, to searching a stream using `grep`. The key difference is that whereas `grep` processes a stream of independent lines, the monitor behaves as though it processes a stream of incrementally-built sequences of events; that is, each

[1]Both are instances of *safety properties* (Schneider, 2000), as the interface expression and the trace are finite.

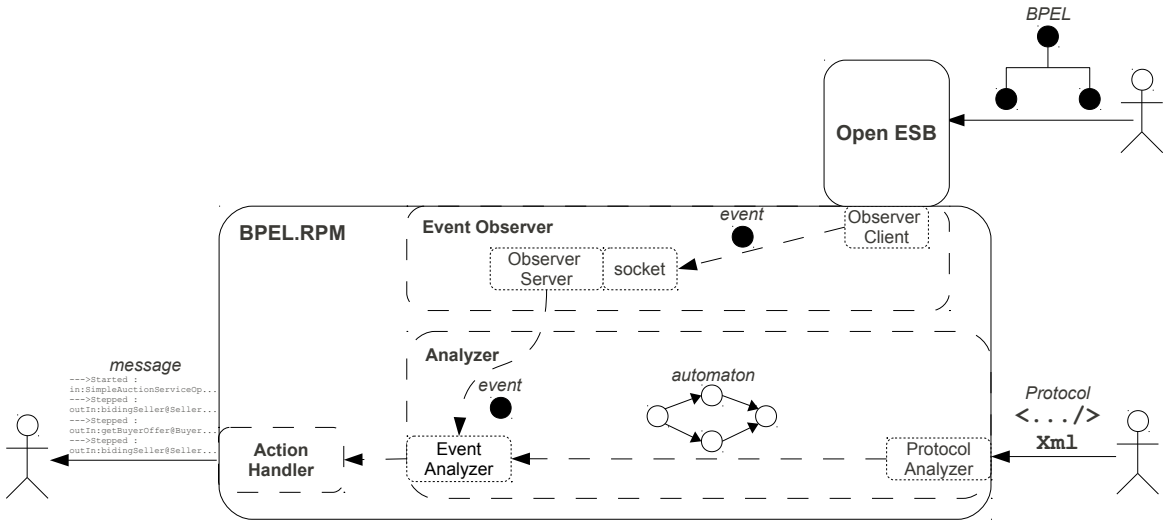

Figure 3: Concrete implementation of BPEL.RPM.

new event is added to the trace, which is then matched against the pattern. Also, when the pattern does not contain an explicit "end of process" marker ("`$`"), an occurrence of the match will be signaled as soon as an appropriate sequence of events is encountered. Thus, an interface expression such as "`a;b;c`" is not exactly the same as "`^.*;a; b;c;.*$`"—in the latter case, a success (as well as a failure) would be signaled only when the monitored process terminates—, whereas they would be equivalent in the `grep` case.

4.2 Event Observer

The event observer component identifies the concrete events from the BPEL process *that might be* of interest, and then transmits those events to the analyzer component. Its implementation depends on the specific BPEL engine being used. In BPEL.RPM, we use BPELSE, the Open ESB (Open Enterprise Service Bus)[2] BPEL Service Engine.

BPELSE allows to dynamically register information about running BPEL process instances. This information is registered in `bpelseDB`, a database associated with the BPEL process engine. To use this database, we defined a *database trigger*, so that whenever information about an event is written in the database, an associated stored procedure is executed. This procedure transmits the information about the event to the monitoring component, using a socket connection, as illustrated in Figure 4, information

[2] `http://open-esb.java.net/`

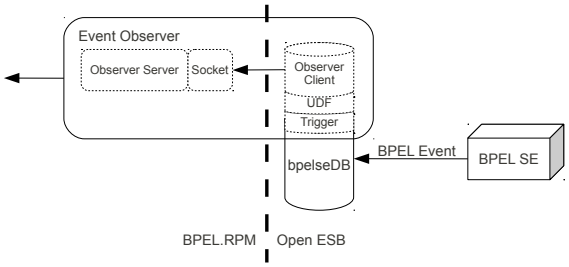

Figure 4: Implementation of BPEL.RPM event observer component.

which is then mapped into an appropriate *BPEL activity*.

4.3 Analyzer

The analyzer is BPEL.RPM's key component. Its role is to keep track of the various events as they occur and to verify the correspondence between those events and the patterns to be detected, as expressed by the interface expressions. To perform this task, the component proceeds as described in the following—Figure 5 shows its overall implementation.

First, the interface expression relative to which a process is to be monitored is transformed into an appropriate finite state automaton. A mapping of the automaton symbols into process activities is also created. These steps are performed by the protocol analyzer (Figure 5). The automata is created using the

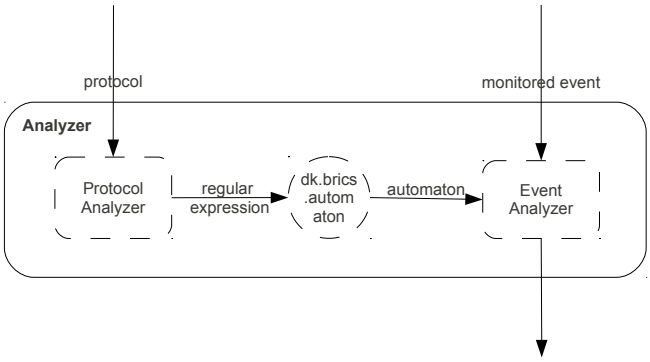

Figure 5: Implementation of BPEL.RPM analyzer.

`dk.brics.automaton`[3] Java package, which provides deterministic as well as non-deterministic automata implementation with support for standard regular expression operations.

Once the automata has been created, event analyzing and monitoring can then begin. Whenever an event occurs, the analyzer proceeds as follows:

- The analyzer retrieves the activity associated with the event;
- If the event activity is not one of the activities to be monitored, the event is simply ignored. Otherwise, if the activity is indeed one of those being monitored, then:
  - The event is added to the execution trace;
  - The state of the automata is advanced according to the received event;
  - Based on the resulting automata state, callbacks to the action handlers are performed.

Various action handlers (i.e., views) can be associated with a specific monitor instance (i.e., model). We use the "Observer Pattern" to decouple the action handlers from the monitor per se, where the required callbacks are performed through an appropriate `MonitorListener` interface.

4.4 Action handler

The run-time action handler component's purpose is to encapsulate a specific reaction strategy with respect to the monitored protocol. We allow four different types of responses (notifications) depending on the state of the protocol:

- When monitoring of the protocol is initiated;
- When an event occurs so that the protocol state must be changed;
- When the protocol successfully reaches completion;
- When the protocol fails.

In our current prototype implementation, we use a simple reaction strategy, namely, we simply log the various events. However, more complex reaction strategies are possible, including interacting with the BPEL engine through the *BPEL Management API*[4].

5 CONCLUSION AND FUTURE WORK

In this paper, we presented a concrete implementation of a run-time monitor for WS-BPEL concrete processes. Monitoring is performed with respect to interface expressions, a form of regular expressions that emphasize externally observable behavior, much like behavior protocols allow for software components (Plasil and Visnovsky, 2002). Such interface expressions can express the overall dynamic interface specification of a process as well as particular sequences of events of interest.

A limitation of interface expressions is that they abstract from the operations' arguments. Thus, all invocations (resp., reception) of a given operation from a specific partner link are represented by the same symbol. As future work, we plan to extend our expressions with message arguments specification.

REFERENCES

Andrews, T., *et al.* (2003). Business process execution language for web services (BPEL4WS) version 1.1. `http://www-128.ibm.com/developerworks/library/ws-bpel`.

Clements, P. and al. (2003). *Documenting Software Architectures: Views and Beyond*. Addison-Wesley.

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

Plasil, F. and Visnovsky, S. (2002). Behavior protocols for software components. *IEEE Tran. Soft. Eng.*, 28(11):1056–1076.

Schneider, F. (2000). Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50.

Tremblay, G. and Chae, J. (2005). Towards specifying contracts and protocols for Web services. In *MCETECH '05*, pages 73–85.

[3] `http://www.brics.dk/automaton`

[4] `http://ode.apache.org/bpel-management-api-specification.html`