A Petri Net based Runtime Monitoring Method for Web Services specified with BPEL

Jun Zhu
School of Computer Science
National University of Defense Technology
Changsha Hunan 410073, China
Email: mail.zhujun@gmail.com

Fabrice Kordon
LIP6 - CNRS UMR 7606
Université Pierre & Marie Curie
4 place Jussieu, 75252 Paris Cedex 05, France
Email: Fabrice.Kordon@lip6.fr

***Abstract*—BPEL (Business Process Execution Language) is one of the dominant ways to specify interactions between Web services. However, it is difficult to deal with behavioral properties of web services. Typically, well defined protocols may be violated by clients, thus leading servers to inconsistent states.**

In this paper, we propose to tackle this problem thanks to an automatically generated runtime monitor from the BPEL specification. First, we extract a web service protocol from its specification. Then we generate a monitor capturing communications from/to the server and detecting inappropriate use of this protocol.

I. Introduction

Context: BPEL [17] is an industry standard proposed by OASIS and supported by major service providers such as IBM, Oracle, Sun, etc. It is one of the dominant service composition description languages for Web Services (WS). BPEL extends the WS interaction model to support business transactions.

The main objective of BPEL is to define protocols (*i.e.* exchange of messages) between several Web services composed to perform advanced functions. In this composition, a single BPEL process needs to converse with other WS (usually called *associated WS*) via ordered interactions based on SOAP (Simple Object Access Protocol) messages. As a result, the composition, as a new single WS, establishes a new *interaction (conversation) protocol*, which corresponds to a new autonomous behavior.

Problem: However, it is difficult to deal with *behavioral properties* of WS compositions. Typically, well defined protocols to request servers may be violated by clients (or associated WS), thus leading servers to inconsistent states. There is no guarantee that clients and associated WS behave as expected because no *conformance* between WS interactions and their specification is guaranteed in current systems.

When a WS runs, protocol messages are the only evidence to check if the interaction protocol corresponds to its specification. This conformance concerns both clients and associated WS. So *runtime monitoring* on such messages is a way to perform conformance checks and evaluate whether interactions comply with the BPEL specification.

Contribution: This paper aims at proposing a solution to check for protocol conformance at runtime. To do so, we automatically generate a runtime monitor from the BEPL specification. The BPEL specification is transformed into a Colored Petri Net [7] (CPN), a formal description technique suitable to capture concurrent behavior. We then generate a runtime monitor capturing communications from/to the WS and using the CPN description to detect inappropriate use of the protocol. Our monitoring approach is *Message Oriented*. The execution of the generated monitor is driven by the captured SOAP messages.

Contents: Section II briefly presents some related work about formal representations of Web service behaviors and monitoring techniques. Section III details the way we transform a BPEL description into Petri Nets and build the associated WS monitor. Finally, Section IV shows, based on experiments, that our approach can be efficiently operated.

II. Related Work

Related work is divided in two parts: formal representation of BPEL specifications and monitoring techniques.

A. Formal representations of BPEL specifications

Several approaches have been introduced to capture the semantics of BPEL by means of formal methods, such as process algebras, automata or Petri nets. The purpose is to verify some properties on the WS composition. We do not aim at providing a full presentation of related work, we only try to identify some typical approaches in the domain.

Process algebras: Process algebra is used as a formal basis to analyze BPEL specifications [9], [5], [19]. [9] express BPLE interaction thanks to Process Algebra. [5] defines a mapping from BPEL to process algebraic LOTOS to verify some temporal properties. [19] uses CCS to describe and validate interactions between Web services.

Automata: Various classes of automata are also used to model BEPL constructions such as plain automata [6] or time automata [4]. The result of the mapping can be used to compute both qualitative and quantitative properties thanks to model checkers. In [6], some LTL properties are verified with SPIN. [4] presents how web services can be translated into a timed automata to be processed with UPPAAL.

Petri Nets: Several studies propose to map BPEL to P/T nets:

- The `BPEL2PN` tool proposes a Petri net-based semantics for BPEL. It deals with standard behaviors and exception handling [10].

- The BPEL2PNML tool focuses on the BPEL control flow constructs. It uses the model to detect unreachable activities and conflicting messages [18].
- The BPEL2oWFN tool uses a special class of Petri nets (open workflow net) to model the interactional behavior of BPEL specification [15]. It checks static analysis requirements available in BPEL such as cyclic control links or illegal access to non initialized variables.

Since we aim at the production of a monitor capable of dealing with several simultaneous sessions, we need colored tokens to differentiate them. Thus, if some of the proposed transformation patterns can be reused, they must be adapted.

Other works deal with CPN. In [24], an interesting approach is presented to verify Web services composition. They rely on the BPEL specification but do not provide precise mapping rules to CPN. However, they identify the important BPEL constructions that capture the behavior of a WS.

Such rules are better defined in [22], [20]. However, they use a specific class of Petri nets: Workflow nets. But, even if several rules can be reused, the authors focus on data flow more than on the control we consider for monitoring (the behavior we consider is the one of the interaction protocol).

Summary: all the presented approaches are relevant to represent the behavior of a system. However, we chose CPN (as in [20]) the three following reasons.

First, CPN are a mathematically founded modeling notation [7], with a large variety of powerful analysis tools.

Second, a CPN based representation of BPEL behavior is much more compact than an automata-based one. This is particularly true because we deal with simultaneous sessions in WS. The number of sessions must be bounded, since implementation does not allow an infinite number of parallel sessions. This is not a major drawback. The Petri net is used to guide the execution and not to generate the full state space; so, there are only a limited number of configurations to store simultaneously (we only deal with a few current states of the system) instead of a complete automaton (or push-down automaton) in most current monitoring approaches [23]. Moreover, since CPN allow us to identify sessions, protocol violations can be unambiguously and precisely identified.

Third, it is known that Petri nets are suitable to express the semantics of business processes [22]. They also proved a strong basis for computer-aided modeling and formal verification, especially in the control part of distributed systems.

Among the various types of Petri nets, it appears that P/T Nets-based approaches generate larger specification. Moreover, we produce a more compact specification of the protocols by using of colored tokens to identify sessions. This also enables a dynamic (but bounded by the color types) management of sessions.

Most current studies producing Petri nets from BPEL generate models that are too complex for monitoring because the proposed patterns contain too many details that are useless for message-oriented monitoring. However, the patterns proposed in [20] are simpler ; we reuse some in this work (section III-A).

B. Monitoring techniques

The community elaborated numerous monitoring tools for WS. We divide them in two categories depending on the way monitor are processed: *offline* and *online* monitoring.

Offline Monitoring: [21] proposes a representative solution of such approaches. Petri nets are derived from BPEL specifications. Traces of SOAP messages (from logs) are used to check conformance of the execution against language associated to the Petri nets.

Such approaches are *post mortem* and thus out of the scope of our work. We want to trigger actions when problems are detected at runtime.

Online Monitoring: Contrary to the previous technique, online monitoring occurs during the execution of web services.

- [3] provides a proxy-based solution to support the execution of monitoring rules at runtime. It uses a method to weave monitoring rules dynamically into the process.
- [13] proposes a monitoring framework for WS interactions. It monitors the interactional behaviors of WS against *pre-defined interaction constraints* that are used to detect protocol violations.
- [2] introduces a monitor specification language to express boolean, statistic and time-related properties of a single BPEL session. The authors propose a solution to translate the specification into Java code to implement the WS monitor; this solution handles one session only.
- [16] proposes a monitoring approach by intercepting events exchanged between the composed processes. The approach can check for violations of behavioral properties and additional monitoring requirements specified in a kind of event calculus.

From the point of view of acquiring information for monitoring, the weaving of monitoring rules ([3]) is very intrusive. It influences the execution of composed WS and decreases the performance of the involved servers. On the contrary, the message catching mechanism used in [16], [12], [13] and [2], is much better in terms of intrusiveness and performance. As a result, our approach adopts a similar solution to intercept SOAP messages as events for monitoring.

From the point of view of monitoring methods, [13] relies on constraints definition, [2] defines a specification language for the monitor, [3] uses self-defined monitoring rules and [16] uses event calculus to describe monitoring requirements. However, the use of such constraints are dedicated to specific violations detections of the involved protocols. So, to provide our monitor with a more accurate view on the protocols to be monitored, we chose to rely on the BPEL specification that is transformed into CPN.

Summary: Due to the limitation of post mortem methods, we do not adopt offline monitoring. We prefer online monitoring since it enables the activation of actions, when protocol violations are detected during the monitored WS execution.

We prefer to use message catching approaches instead of code instrumentation methods in order to reuse WS sources “as is”. This code is transformed into a CPN that represents the protocol reference.

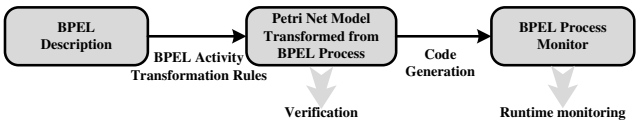


Figure 1. Overview of the Approach

III. Building a Monitor from a BEPL description

This section details how we build a monitor from a BPEL description. This process is divided in two steps presented in figure 1: the transformation of a BPEL specification into a CPN and then the generation of a monitor embedding this CPN to be used as a description of the reference protocol. Let us note that the produced CPN can also be used for verification purpose.

A. BEPL to Petri Net

CPN Model for Composed Service Process: let us first informally define CPN (more details can be found in [7]). This definition introduces entities suitable for the description of WS.

Definition 1 (CPN): A CPN N is a tuple (P, T, F, C), where:

1) P is a finite set of places; T is a finite set of transitions ($P \cap T = \emptyset$); F is a finite set of arcs ($F \subseteq (P \times T) \cup (T \times P)$); C is a finite set of colors.
2) $P = P_I \cup P_M$ with $P_I \cap P_M = \emptyset$, where P_I is a set of `internal` places and P_M is a set of `message` places, corresponding to the connection messages in process. This partition separates places for the control flow of the protocol (P_I) from the message flow (P_M).
3) C is the color set deduced from the WS sessions (each value represents one interaction with a client or an associated Ws). This allows to identify sessions, each one executing the protocol (described with BPEL).
4) $F = F_I \cup F_M$ with $F_I \cap F_M = \emptyset$, where $F_I \subseteq (P_I \times T) \cup (T \times P_I)$ is a set of arcs and $F_M = F_M^R \cup F_M^S$ where $F_M^R \subseteq (P_M \times T)$ represents incoming messages arcs and $F_M^S \subseteq (T \times P_M)$ represents outgoing messages arcs.

We note $\bullet p \in T$ (resp. $\bullet t \in P$) the predecessors of place p (resp. transition t) and $p\bullet \in T$ (resp. $t\bullet \in P$) the successors of place p (resp. transition t).

Transformation Rules: as mentioned in section II-A, we reuse some interesting patterns from [20] with some adaptations, such as patterns *receive*, *reply*, *sequence* etc. Due to lack of space, only two typical BPEL activities (one basic activity – *invoke* – and one structured activity – *switch*) are presented. They show how to map BPEL constructions to a CPN pattern. Other patterns are defined in a similar way. Each presented rule has a name, a precondition to be satisfied to operate the rule, and defines a mapping to CPN. Let us note that each CPN pattern has two special places: one source place p_f (where $p_f \in P_I$ with $\bullet p_f = \emptyset$) and one sink place p_l (where $p_l \in P_I$ with $p_l\bullet = \emptyset$).

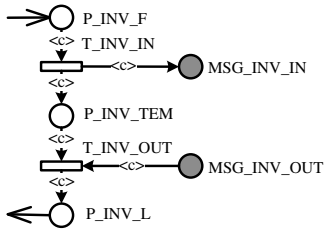
Name	**Invoke**
Precondition	Parsing of a <invoke> lexeme with two variables; `inputVariable` and `outputVariable`.
Mapping to CPN	

Figure 2. Transformation Rule for the BPEL Basic Activity *Invoke*

The rule presented in figure 2 transforms a BPEL basic activity *invoke* (request-response) into a CPN. It is activated when an <invoke> with `inputVariable` and `outputVariable` is parsed. The outgoing request message of the corresponding session ($c \in C$) is dropped in place `MSG_INV_IN` by the transition `T_INV_IN`. The return message is consumed by transition `T_INV_OUT` fromq place `MSG_INV_OUT`. `P_INV_F` is the source place of the pattern and `P_INV_L` is the sink place.

When the invoke is one-way, the produced CPN contains only the request sending. This is handled by a variation of this rule dedicated to this particular case.

The rule presented in figure 3 transforms a BPEL structured activity *switch* into a CPN. It is activated when a <switch> with at least one `case` alternative (and possibly an optional element `otherwise`) is parsed. Each alternative is mapped to a dedicated conditional branch. The optional `otherwise` element is mapped to a default branch (if there are no satisfied conditions, the `otherwise` branch is chosen thanks to the associated guard computed from the negation of all other guards). `BLOCK` frames stand for the model deduced from the instructions of the associated alternative.

Concatenation of CPN Patterns: since all transformed patterns begin and end with places (source place and sink place), composition can be achieved by place fusion.

Figure 4 explains how two CPN patterns derived from two successive BPEL instructions are composed. We provide an example in figure 5 where the rule is activated for an `invoke`

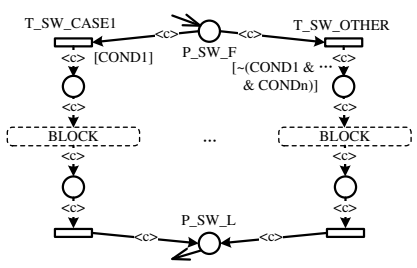
Name	**Switch**
Precondition	Parsing of a <switch> lexeme with at least one `case`.
Mapping to CPN	

Figure 3. Transformation Rule for the BPEL Structured Activity *Switch*

Name	Concatenation of patterns
Precondition	Two given CPN patterns issued from two successive BPEL activities: $N_1 = (P_1, T_1, F_1, C)$ and $N_2 = (P_2, T_2, F_2, C)$.
Effect on CPN	Concatenated CPN $N = (P, T, F, C) = N_1 + N_2$ where: 1) source place of N_2 is merged with sink place of N_1, 2) $P = P_1 \cup P_2 \setminus \{p_{1l}\}$ where p_{1l} is the sink place of N_1 (fusioned with the source place P_{2f} of N_2), 3) $T = T_1 \cup T_2$, 4) The composed arc set $F = (F_1 \cup F_2 \cup F_A) \setminus F_D$ where $F_A = (\bullet p_{1l} \times \{p_{2f}\})$ represents the created arcs from predecessors of place p_{1l} to place p_{2f} and $F_D = (\bullet p_{1l} \times \{p_{1l}\})$ represents the deleted arcs associated to the deleted place p_{1l}. Remark: the color set C remains the same all over the CPN patterns produced from the whole BPEL specification.

Figure 4. Transformation Rule of Pattern Concatenation

followed by a `switch`.

Reducing the CPN complexity: our composition mechanism may lead to large models when BPEL specifications are complex. To avoid this, we apply the CPN reduction rules defined in [8]. They reduce the complexity of the resulting CPN by suppressing objects without changing its behavior. Of course, transitions and places corresponding to monitored events (like message reception or emission) cannot be suppressed.

B. Architecture of our WS Monitor

The architecture of our WS monitor is depicted in figure 6. We identify two major components: the message catching mechanism (A) and the monitoring module (B).

Message Catching Mechanism: the message catching mechanism captures incoming and outgoing SOAP messages and passes them to the monitor module that checks if the message is valid or not against the protocol model expressed with a CPN.

To capture SOAP messages in real time, we rely on the underlying communication mechanisms. For this experiment, we chose Apache ODE and AXIS2 [1] because they are two well-known and widely used open source execution environments for WS. Their flexibility allowed us to plug our approach for experimentation purpose.

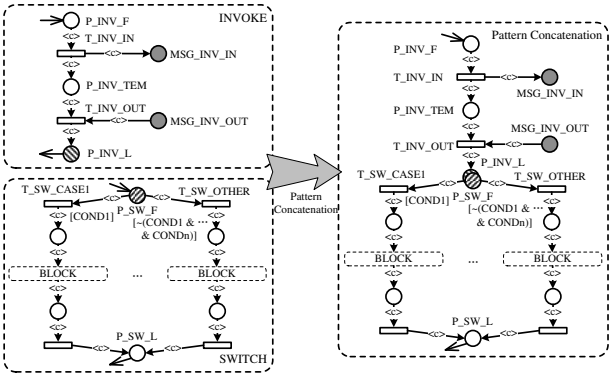


Figure 5. Illustration of CPN concatenation

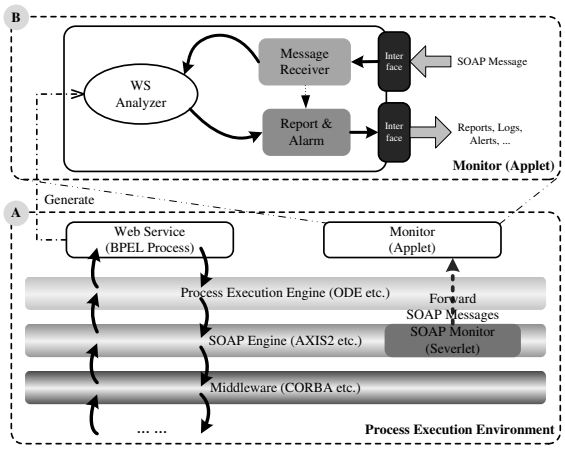


Figure 6. Entire Architecture of WS Monitor

Monitoring module: the monitoring module includes three components: the Message Receiver (MR), the WS Analyzer (WA) and the Reports & Alerts (RA).

The MR encapsulates the message catching mechanism and ensures independence from its implementation. It buffers messages and delivers them to the WA in the same order as they were received.

The WA is generated from the CPN model. It is executed and synchronized with the caught messages to set up, thread per thread, the evolution of the WS protocol. If no transition can be fired in the CPN, the protocol is violated. This event is then passed to the RA.

The RA is notified when a problem occurs and handles it accordingly to its configuration. The security manager can set up actions to be taken such as reporting in a log, executing some dedicated code, filtering requests from the machine executing the involved client or just stopping the WS.

The monitoring module gets all caught messages and checks if they correspond to a correct state in the WS protocol. It follows the algorithm depicted in figure 7. The initialization consists in initializing variables such as the current state of WS sessions (usually handled by one dedicated thread), mapping of WS sessions to tokens in the CPN, message queue, etc.

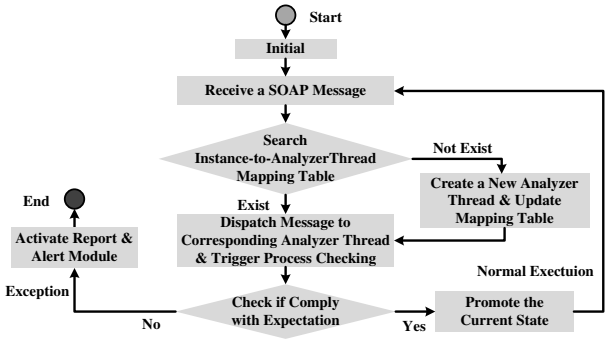


Figure 7. Algorithm of the monitoring module

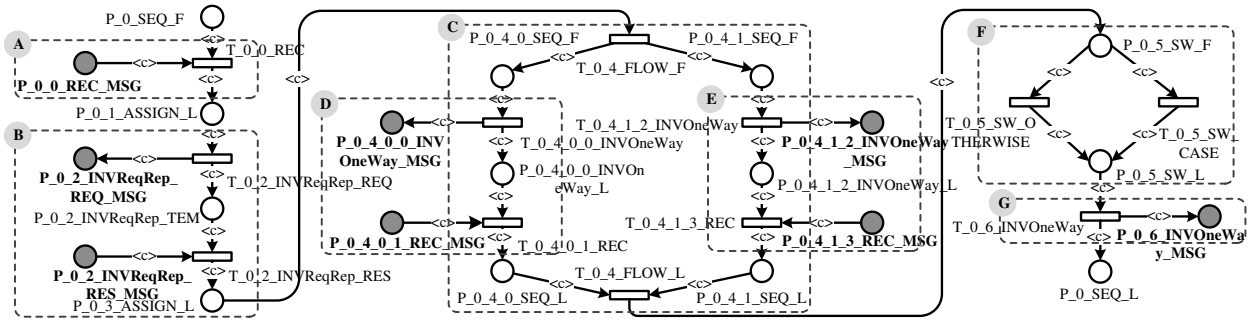


Figure 8. CPN model generated from the ETA example

Then SOAP messages coming from the catching mechanism are treated one by one. Each message is associated with a session that is mapped to a token. So, when a message is processed, it is mapped to the associated token. If no transition is enabled for this token, then the WA module detects a problem. The algorithm is detailed in the next subsection.

C. Generating the WS analyzer

The WS analyzer module is automatically generated from the CPN model and thus is a rigorous representation of the BPEL behavior. The other modules of the WS monitor are reusable libraries. Code generation is done in three phases:

1) *Collect statistic information about model*: this information (cardinality of P_I, P_M, T, etc.) is used to size data structure and predefine the mapping of messages.
2) *Code generation*: the WA is generated according to the control flows defined in the CPN detailed below.
3) *Module integration*: the MR and RA modules are linked to the WA. The result can be deployed in the AXIS2 WS execution environment where we chose to plug the message catching system.

Generated code: each transition $t \in T$ is mapped to a constant and corresponds to `case` in a `switch` instruction where the corresponding code is inserted. We proceed similarly with places $p \in P_M$. The `switch` is located in a `loop` and implements the control flow deduced from the BPEL specification. This code is executed by a dedicated thread associated with the associated session.

The current state of a session is stored in a dedicated variable (the maximum number of sessions must be configured for the WS). This variable is updated once the code associated to places and transitions is executed.

A message mapping function is produced from P_M and allows the WA to interact with the message catching mechanism.

IV. Experimentations

This section illustrates our approach on several examples. These examples are treated with our prototype implementation based on the Coloane [14] Eclipse plug-in. Coloane is a free software generic graphic editor.

A. The Employee Travel Arrangements example

Let us first use the service Employee Travel Arrangements (ETA) defined in [11]. It is a classical example of WS composition (see figure 9). It contains three typical structured activities `flow`, `switch` and `sequence`, and three basic activities `receive`, `assign` and `invoke`. It also manages three associated WS (*employeeTravelStatus*, *AmericanAirlines* and *DeltaAirlines*). It deals with 8 SOAP messages.

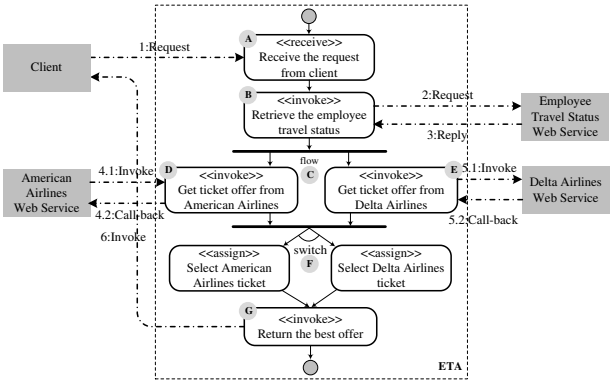


Figure 9. The BPEL specification of the ETA example

The Generated Petri Net: the resulted CPN is presented in figure 8. It has 21 places, 12 transitions and 34 arcs and is produced in less than 2 seconds. Dashed-line frames outline major portions of this net, to make a connection with the corresponding BPEL activities. These frames are labeled to correspond to the elements outlined in figure 9. Gray places correspond to SOAP messages.

The Generated Code: The generated WS monitor for this example is 1347 LOC in Java. Among these, the WA is 118 LOC, the MR is 165 and the RA is 206. Code for MR and RA remains constant.

Comparison to Other Tools: We also run our tool on two other examples extracted from the community: the Loan Approval Process (LAP) and the House Loan of Bank (HLB) [17]. We processed these three examples with BPEL2oWFN[1] [15] and BPEL2PNML [18]. The objective

[1]Development of BPEL2PN stopped in 2005. BPEL2oWFN, its successor, supports at least all its features.

Tools	Place	Transition	Arc	Code for WA
Example ETA (154 BPEL LOC) 17 Total Activity (11 Basic + 6 Structured) 8 Messages				
BPEL2PNML	118	116	313	-
BPEL2oWFN	65	84	249	-
Our tool	21	12	34	118 LOC
Example LAP (98 BPEL LOC) 8 Total Activity (6 Basic + 2 Structured) 4 Messages				
BPEL2PNML	57	46	146	-
BPEL2oWFN	74	87	280	-
Our tool	19	8	30	91 LOC
Example HLB (186 BPEL LOC) 30 Total Activity (17 Basic + 13 Structured) 12 Messages				
BPEL2PNML	202	195	444	-
BPEL2oWFN	143	188	542	-
Our tool	50	28	84	394 LOC

TABLE I
STATISTICS ON GENERATED CPN AND MONITOR

Rate of "bad" Clients	Detected Violations	Rate of "bad" Clients	Detected Violations
5%	100%	30%	100%
10%	100%	40%	100%
20%	100%	50%	100%

TABLE II
EVALUATING PROTOCOL VIOLATION DETECTION RATE

is to compare the size of the generated Petri nets (no code generation is offered by BPEL2PNML and BPEL2oWFN). When available, all possible optimizations of these tools were activated.

Table I compares the produced CPN with ours (light gray lines). It shows that our patterns compete with other ones and lead to smaller CPN. This is probably because our patterns rely on the control flow of the WS protocol.

Let us note that the WA generated for these examples is also quite small as the last column shows. It allows a small memory footprint and does not overload the execution of the web service.

B. Evaluating Monitor Efficiency

We first evaluate if protocol violations are appropriately detected by our monitor. To do so, we run the WS and its associated monitor against "good" and "bad" clients. Good clients just behave correctly while bad clients generate random violations at various stages of the supported protocol.

This experiment is reported in table II for a rate of "bad" clients varying from 5% to 50%. We then check if the number of detected violations corresponds to the number of expected one. As the table shows, 100% of the violations were detected and no false negative was observed. These experiments were evaluated on 10^5 executions of the WS.

C. Performance Evaluation

One key feature of program monitoring is non-intrusiveness: the monitor must be efficient in terms of memory and execution. Experimentation were done on two machines with a Core™ 2 Duo at 3GHz, 2GB RAM and running Mandriva Release 2009.0 (Linux kernel 2.6.27.14). The WS execution environment is Java SE (build 1.6.0_15-b03), Apache Tomcat 6.0.20 and Apache ODE 1.3.3. We used Apache Jmeter 2.3.4 to measure the WS response time and the `pmap` Unix command to evaluate memory footprint.

Memory footprint: it is quite small and never exceeded 6.47MB during benchmarks (this was measured for the monitoring of numerous parallel session in the experiment summarized in figure 11). This is due to the small size of the code and the fact that only a very few markings have to be maintained by the WA. The number of maintained markings is strongly related to the maximum number of WS sessions to be handled by the system and thus. Thus, an appropriate configuration of the system allows to control the size of required memory.

Execution Overhead for sequential execution: let us first measure the execution overhead when the WS processes sessions sequentially. To do so, we compare a set of WS executions with monitoring to the same execution without monitoring. To evaluate potential variations on the impact of monitoring when realization of services require CPU, we introduce specific code that uses between 8 and 9 ms of CPU and a variable *code loop* that corresponds to the number of time this code is executed when a SOAP message is processed.

Figure 10 shows the response time measured by Jmeter. It shows that, whatever the process time of a SOAP message is, the overhead does not seem to be impacted. The observed overhead is about 5.5%. Here, we extracted our measure from the analysis of 10^5 SOAP messages for each value of *code loop*.

Execution Overhead for parallel execution: to assess that our approach also stands for simultaneous executions of WS, we evaluate the response time when numerous clients simultaneously contact the WS we used for our sequential execution measures. In that experiment, *code loop* $= 3$. The number of processed SOAP messages is then 2×10^5 and requests are shared by the clients (*i.e.* 10 simultaneous clients make 2×10^4 requests each and 100 simultaneous clients make 2×10^3 requests each).

Performances are presented in figure 11. It shows that the management of parallel sessions has an impact on the WS performances (this comes from the WS execution environment). It also shows that the monitor overhead also increases with the number of simultaneous sessions. This is due to the

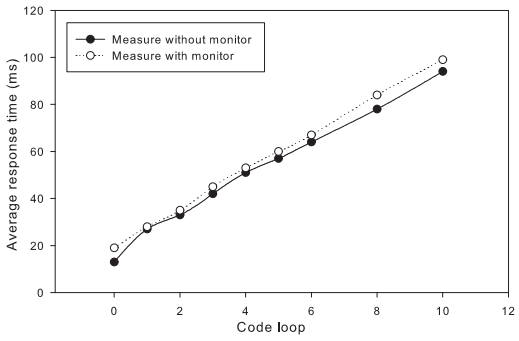


Figure 10. Average response time for various execution time of the WS

management of shared data structures required to map sessions to colored tokens. The ratio between the two curves however remains quite acceptable.

V. Conclusion

This paper presented a quickly operable solution to automatically produce Web services (WS) monitors from BPEL specifications. To do so, we extract the protocol defined in the BPEL specification and encode it by means of Colored Petri Nets (CPN). These are then used as a compact way to describe this protocol in order to detect violations from clients or associated WS. Monitoring relies on the capture of incoming and outgoing communications. They provoke a tentative execution of the CPN: if no execution is possible, then a violation is detected.

Our approach has been implemented and monitors have been generated from typical BPEL examples. The size of the produced monitor is reasonable and can be embedded on small servers.

Experimentation showed that both memory footprint and execution overhead are reasonable, even when numerous WS sessions are simultaneously handled. This shows the viability of such a solution to monitor heavily used servers.

Our solution also provides another advantage not exploited in this paper. The CPN model can serve as a basis for formal verification purpose as it is the case in similar work [15].

Future Work: so far, our technique runs in parallel of the WS. And thus, actions remain limited to logging or stopping the WS execution, when protocol violations are detected. Further work will thus focus on a closer integration of the monitor to the WS in order to have a better control of the WS when problems are detected.

Acknowledgment

The authors would like to thank their research sponsors the National Natural Science Foundation of China (Grant 90818028), the National High Technology Development 863 Program of China (Grant 2007AA010301), the China Scholarship Council and the hosting from Université P. & M. Curie.

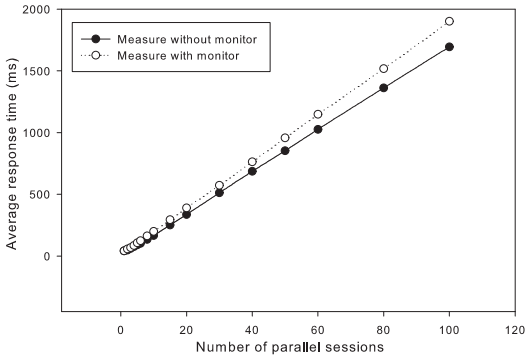


Figure 11. Impact of parallelism on the monitor overhead

References

[1] Apache. Apache Axis2 User's Guide, http://ws.apache.org/axis2/1_5_1/userguide.html, 2008.

[2] F. Barbon, P. Traverso, M. Pistore, and M. Trainotti. Run-Time Monitoring of Instances and Classes of Web Service Compositions. In *4th IEEE International Conference on Web Services (ICWS'06)*, pages 63–71, 2006.

[3] L. Baresi and S. Guinea. Towards Dynamic Monitoring of WS-BPEL Processes. In *3rd International Conference on Service-Oriented Computing (ICSOC'05)*, pages 269–282, 2005.

[4] G. Diaz, J. Pardo, M. Cambronero, V. Valero, and F. Cuartero. Automatic Translation of WS-CDL Choreographies to Timed Automata. In *2nd International Workshop on Web Services and Formal Methods (WS-FM'05)*, volume 3670, pages 230–242. Springer, 2005.

[5] A. Ferrara. Web Services: a Process Algebra Approach. In *2nd International Conference on Service Oriented Computing (ICSOC'04)*, pages 242–251, 2004.

[6] X. Fu, T. Bultan, and J. Su. Analysis of Interacting BPEL Web Services. In *13th International Conference on World Wide Web (WWW'04)*, pages 621–630, 2004.

[7] C. Girault and R. Valk. *Petri Nets for Systems Engineering: A Guide to Modeling, Verification, and Applications*. Springer Verlag, 2003.

[8] S. Haddad. A reduction theory for coloured nets. *Advances in Petri Nets*, 424:209–235, 1989.

[9] S. Haddad, T. Melliti, P. Moreaux, and S. Rampacek. Modelling web services interoperability. In *6th International Conference on Enterprise Information Systems (ICEIS)*, pages 287–295, 2004.

[10] S. Hinz, K. Schmidt, and C. Stahl. Transforming BPEL to Petri Nets. In *3rd International Conference on Business Process Management (BPM'05)*, pages 220–235, 2005.

[11] M. B. Juric. A Hands-on Introduction to BPEL, http://www.oracle.com/technology/global/en/pub/articles/matjaz_bpel1.html, 2008.

[12] Z. Li, J. Han, and Y. Jin. Pattern-Based Specification and Validation of Web Services Interaction Properties. In *3th International Conference on Service-Oriented Computing (ICSOC'05)*, pages 73–86. Springer, 2005.

[13] Z. Li, Y. Jin, and J. Han. A Runtime Monitoring and Validation Framework for Web Service Interactions. In *2006 Australian Software Engineering Conference (ASWEC'06)*, pages 70–79, 2006.

[14] LIP6/MoVe. Coloane web page, http://move.lip6.fr/software/COLOANE.

[15] N. Lohmann, P. Massuthe, C. Stahl, and D. Weinberg. Analyzing Interacting WS-BPEL Processes using Flexible Model Generation. *Data & Knowledge Engineering*, 64(1):38–54, 2008.

[16] K. Mahbub and G. Spanoudakis. Run-Time Monitoring of Requirements for Systems Composed of Web-Services: Initial Implementation and Evaluation Experience. In *3rd International Conference on Web Services (ICWS'05)*, pages 257–265, 2005.

[17] OASIS. Web Services Business Process Execution Language Version 2.0 (WS-BPEL 2.0), April 2007.

[18] C. Ouyang, E. Verbeek, W. van der Aalst, S. Breutel, M. Dumas, and A. ter Hofstede. Formal Semantics and Analysis of Control Flow in WS-BPEL. *Science of Computer Programming*, 67(2-3):162–198, 2007.

[19] G. Salaun, L. Bordeaux, and M. Schaerf. Describing and Reasoning on Web Services using Process Algebra. *International Journal of Business Process Integration and Management*, 1(2):116–128, 2006.

[20] W. Tan, Y. Fan, and M. Zhou. A Petri Net-Based Method for Compatibility Analysis and Composition of Web Services in Business Process Execution Language. *IEEE Transactions on Automation Science and Engineering*, 6(1):94–106, 2009.

[21] W. M. P. van der Aalst, M. Dumas, C. Ouyang, A. Rozinat, and E. Verbeek. Conformance Checking of Service Behavior. *ACM Transactions on Internet Technology (TOIT)*, 8(3):1–30, 2008.

[22] H. Verbeek and W. van der Aalst. Analyzing BPEL processes using Petri nets. In *2nd International Workshop on Applications of Petri Nets to Coordination, Workflow and Business Process Management*, pages 59–78, 2005.

[23] J.-B. Voron and F. Kordon. Evinrude: A Tool to Automatically Transform Program's Sources into Petri Nets. *Petri Net Newsletter*, 75:19–38, 2008.

[24] Y. Yang, Q. Tan, and Y. Xiao. Verifying Web Services Composition Based on Hierarchical Colored Petri Nets. In *1st International Workshop on Interoperability of Heterogeneous Information Systems (IHIS'05)*, pages 47–54, 2005.