Huang T, Wu GQ, Wei J. Runtime monitoring composite Web services through stateful aspect extension. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 24(2): 294–308 Mar. 2009

Runtime Monitoring Composite Web Services Through Stateful Aspect Extension

Tao Huang[1] (黄 涛), *Member*, *CCF*, Guo-Quan Wu[1,2] (吴国全), and Jun Wei[1] (魏 峻), *Member*, *CCF*

[1] *Institute of Software, Chinese Academy of Sciences, Beijing 100190, China*

[2] *Department of Computer Science and Technology, University of Science and Technology of China, Hefei 230027, China*

E-mail: {tao, gqwu, wj}@otcaix.iscas.ac.cn

Received March 17, 2008; revised December 21, 2008.

Abstract The execution of composite Web services with WS-BPEL relies on externally autonomous Web services. This implies the need to constantly monitor the running behavior of the involved parties. Moreover, monitoring the execution of composite Web services for particular patterns is critical to enhance the reliability of the processes. In this paper, we propose an aspect-oriented framework as a solution to provide monitoring and recovery support for composite Web services. In particular, this framework includes 1) a stateful aspect based template, where history-based *pointcut* specifies *patterns* of interest cannot be violated within a *range*, while *advice* specifies the associated recovery action; 2) a tool support for runtime monitoring and recovery based on aspect-oriented execution environment. Our experiments indicate that the proposed monitoring approach incurs minimal overhead and is efficient.

Keywords runtime monitoring, aspect-oriented programming, WS-BPEL

1 Introduction

Modern software architectures are increasing dynamic. Among them, Service-Oriented Architecture (SOA) has been emerging as a most promising architectural paradigm which provides support to dynamically evolving software architecture. In this paradigm, individual service providers develop their Web services, and publish them at service registries. Service consumers can then discover the required services from the service registries and compose them to create new services. Web services composition is becoming a promising approach to developing flexible distributed applications in the Internet era. Organizations are increasingly using composite Web services to automate business processes by dynamically selecting and assembling a set of autonomous and loosely coupled Web services. WS-BPEL[1] now represents the de-facto standard for Web services composition. In WS-BPEL, the composition is called process and the composed services are called partner services or partners.

The runtime monitoring of software has strong motivation[2]. In the case of SOA, since publication and discovery may be performed at runtime, binding may also be established and modified dynamically. This high degree of dynamism, while providing great benefits in terms of flexibility, has a severe impact on the system's correctness and on the way verification can be performed. Traditionally, verification of correctness is performed statically, based on the known components that compose the application. For the service-based applications which are often developed by composing services that are made available by third parties and may change without notification, their running behavior may not comply with specified behavior properties[3]. Moreover, for the demands of high level of flexibility, composite Web services need to adapt themselves to rapidly changing environment. Some properties validated at design time may be violated at runtime, which makes runtime monitoring and verification more necessary. Also, monitoring the execution behavior of composite Web services is critical to enhance the reliability of the processes. There is an increasing interest towards self-healing business process[4], which can detect faults and errors instantly and then contain their effects within defined boundaries. Thus it allows applications to recover from the negative effects of such anomalies.

For some intuition about the type of monitoring that composite Web services may require, consider an

Regular Paper

This work is supported by the National Natural Science Foundation of China under Grant Nos. 60673112, 90718033, the National Basic Research 973 Program of China under Grant No. 2009CB320704, and the High-Tech Research and Development 863 Program of China under Grand Nos. 2006AA01Z19B, 2007AA010301.

application of online travel arrangement (OTA) process. Monitoring of such process execution may allow the manager to detect faults and guarantee correct execution. For instance, it can be specified that before a user request to cancel a trip is accepted by the process, both hotel and flight reservation must have been canceled successfully. Similarly, when the failure times of a service invocation pass a certain threshold, it should allow business process to switch to a backup service automatically. In general, business process monitoring encompasses the tracking of particular patterns during the process execution. In case the specified patterns are violated, suitable actions should be enacted timely to supervise the running processes.

Current Web services specification language and orchestration engines offer limited support to accommodate monitoring requirements. The exception handling mechanisms offered by the process orchestration engines do not provide sufficient support for monitoring to detect and handle the broad range of business exceptions or faults that may occur during the process execution. Moreover, the monitoring logic is often scattered across different modules and tangled with the functional specification of the normal process flow. This negatively impacts the maintainability and increases the design complexity.

Aspect-oriented programming (AOP) offers a new set of language features to increase modularity and separation of concerns, which makes it very suitable *for* runtime monitoring[5,6]. However, monitoring composite Web services based on AOP technology is not yet adequately supported in existing work. In most aspect extension to WS-BPEL[5,7,8], the observation that an aspect can make are confined to the current process state: it is not possible to directly observe the history (or the event trace) of the process execution. "Stateful Aspect"[9,10], the advanced feature in Aspect-Oriented Software Development (AOSD), is not supported currently, which makes it difficult to express complex monitoring tasks, such as the behavior constraints mentioned above in the OTA process. There is an extra concern that depends on history tracking, and without language support, will be tangled with the base process and the concern that depends on this history tracking.

This paper presents aspect-oriented runtime monitoring and recovery solution for composite Web services. Specifically, we provide a template that embodies the AOP approach to specify the monitoring requirements and the associated recovery action. The template consists of ⟨*pointcut*, *advice*⟩ pair, where the *pointcut* specifies the behavior constraints, while the *advice* defines what the recovery logic is. Different from existing AOP extension, in our approach, pointcuts are a kind of dynamic quantification[11] and expressed as "*pattern* of interest cannot be violated within a *range*", based on the events occurred during the process execution. In this paper, and the prototype we developed, we assume the composite Web services are implemented in WS-BPEL, the advices are therefore specified in WS-BPEL for consistency with the target implementation. However, as WS-BPEL supports only limited fault handling mechanism, we provide some extended actions, such as *retry*, *alternate* and *skip*. These actions can be combined with normal WS-BPEL constructs to support complex fault handling logic.

In fact, as we will see, the proposed pointcut depends on the events occurred during the process execution, runtime weaving is more suited than static weaving done at the code level. We present a prototype of an aspect-oriented execution environment to support pattern monitoring and advice enforcement. We show how the stateful aspect based template can be efficiently implemented based on finite state automata (FSA) in order to transparently weave history tracking code with the WS-BPEL process at runtime. Some optimization technique is also discussed to speed up the computation, by pruning redundant monitoring based on the analysis of the pointcut. Finally, we describe experiments that indicate the resulting monitoring is efficient and incurs very minimal overhead. All of the above ingredients of the solution correspond to the contribution of this paper.

The remainder of this paper is organized as follows. Section 2 further details stateful aspects and shows three approaches that implement this concept. Section 3 introduces the WS-BPEL language and gives an Online Travel Arrangement business process as a motivation example. Section 4 elaborates stateful aspect based template for the specification of behavior constraints and recovery actions. Section 5 illustrates an aspect-oriented prototype environment, the dynamic weaving mechanism and the corresponding optimization technique. Section 6 describes the experiments to measure the performance overhead of the system. Section 7 discusses the proposed history-based pointcut. Section 8 introduces related work. Finally, Section 9 concludes the paper and provides some directions for further work on this subject.

2 Stateful Aspect

Aspect-oriented programming offers a new set of language features to increase modularity and separation of concerns. In AOP programming, pointcut is predicate that identifies sets of related join points in the execution of a program, where to execute behavior pertaining

to crosscutting concerns.

The essence of AOP programming is making quantified programmatic assertions over programs that otherwise are not annotated to receive these assertions[11]. Quantification implies matching a predicate about a program and it can be distinguished between static quantification and dynamic quantification. Static quantification works over the structure of the program. Dynamic quantification speaks to matching against events that occur in the course of the program execution.

Most common aspect implementation technique (e.g., AspectJ, JBoss AOP) uses static quantification, that is, the pointcut is restricted to describe joinpoint and condition that relate to the current state. Dynamic quantification supports a richer pointcut notation that refers to the history of a computation as a trace of the joinpoints encountered so far. Aspects that depend on the history of program execution are called "stateful aspects". This technique is available in the JAsCo aspect language, and has been recently introduced in an extension to AspectJ, the so-called Tracematches.

JAsCo[10] aims to combine technique from AOP with component-based software development. The extension of JAsCo can specify protocols as a pointcut description. The proposed pointcut language is equivalent to a finite state machine. Transitions in the protocol correspond to runtime events and are determined by a standard pointcut. When the pointcut is matched at runtime, the transition is fired. JAsCo allows the programmer to attach advice to all transitions in the protocol and the advice will be executed as soon as the part of the protocol has been reached. Extra feature supported by the stateful aspects in JAsCo is strict protocols, which only matches if no other joinpoints can be matched between the activation of the transition in the protocol.

Tracematches[9] is an extension of AspectJ for defining stateful aspects. This extension enables the programmer to trigger the execution of extra code by specifying a regular pattern of events in a computation trace. One of the compelling features of Tracematches is the introduction of free variables in the matching patterns and supports to keep traces based on variable bindings. For a regular pattern, different traces can be kept depending on different variable bindings. When a new joinpoint is encountered, only the trace that corresponds to the variable of the joinpoint is updated.

While initial research on AOP has concentrated on the area of object-oriented software development, there are some aspect oriented extensions to the process-oriented composition languages such as WS-BPEL. To support advanced AOSD feature, Padus[12] proposes a stateful aspect extension to BPEL language, where history-based pointcut is specified with the regular expression based on their logic-based pointcut language. To transparently weave history tracking code in the process, Padus adopts static weaving strategy based on finite state automata.

3 Motivation Example

WS-BPEL is essentially a high level specification language with an XML-syntax that describes the execution flow of a process and interaction with other processes. WS-BPEL defines fifteen activity types, among them the most important are the following:

Process ::= *Process* (Activity$_1$, ..., Activity$_n$; V),
Activity ::= BasicActivity | StructuredActivity,
BasicActivity ::= *receive* | *invoke* | *reply* | *assign* | *throw* | *terminate* | *compensate* | *wait* | *empty*,
StructuredActivity ::= *sequence* | *switch* | *flow* | *while* | *pick* | *scope*.

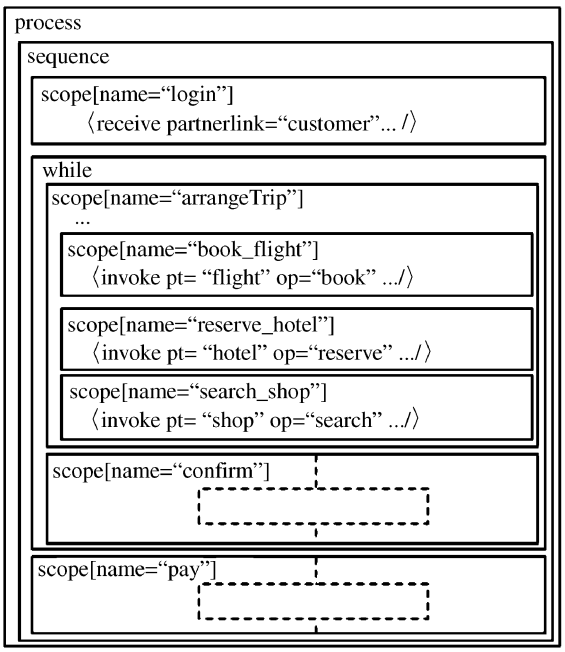

Fig.1. OTA business process.

As a motivation example, consider an online travel arrangement (OTA) process (see Fig.1), which makes a travel plan for the clients. When the agency receives a request from a client, it contacts the flight service. If available tickets are found, the flight service returns an offer including the cost. Similarly, the agency contacts the hotel service and asks for hotel room reservation. The travel agency also offers a shop search service, which charges low fee and returns some sale information near the hotel. After the travel package is available, the

agency prepares and sends the client an aggregated offer which includes transportation, accommodation and sale information. If the client is satisfied with this offer, then she/he sends payment information (e.g., the credit card number) to the agency, which starts the payment procedure. If the payment is successful, the flight and hotel services will emit electronic tickets that the agency will forward to the client.

In the OTA process scenario, we give the following examples to illustrate the features available for monitoring. Possible recovery actions are also described if the constraints are violated.

Example 1. Check whether the current process execution preserves the specified behavior properties. It can be specified that, before a user's request to cancel the trip has been accepted by BP, both hotel and flight reservations must have been canceled successfully. If the non-compliance is detected, some completed activities should be compensated and then notify the process manager.

Example 2. It is possible to monitor the malicious behavior from some users. It can be restricted that the total times of cancelling a trip cannot exceed 4. In case of violation, the process should be terminated timely.

As the WS-BPEL processes are loosely-coupled, they are susceptible to a wide variety of faults, especially those from autonomous partner services. Examples 3 and 4 specify the behavior constraints of the partner services.

Example 3. Consider the behavior of the hotel service in case of a cancellation request. This request can be accepted or refused by the hotel service. However, from the service level agreement between the OTA and the hotel service, the hotel cancellation should be granted whenever the payment of the requested room reservation has not yet been done. If this constraint is violated at runtime, a retry or alternate action (the hotel service is substituted with a backup service) can be performed to achieve user's travelling requirements.

Example 4. To maintain a whole acceptable response time of the OTA process, it can be specified that the execution time of hotel reservation service cannot exceed 5 minutes. Otherwise, the shop search service can be skipped.

4 Stateful Aspect Based Template for Process Monitoring

While AOP seems to be the obvious choice to tackle the monitoring requirements mentioned above, none of the existing aspect oriented systems provides explicit support for some of essential elements, in particular, to express advance patterns of the process execution history. We present a new stateful aspect based template while allows such crosscutting concerns to be expressed concisely. Specially, it allows users to declaratively define the patterns of interest through history-based pointcut and specify associated recovery action to supervise the running process.

Fig.2 shows the stateful aspect based template of Example 1. In the following we detail this template structure: we first introduce history-based pointcut and discuss the rationale for their design. Then the semantics of the proposed pointcut is given. Finally, we present the advices and elaborate some extended fault handling action.

4.1 Join Point and History-Based Pointcut

For WS-BPEL language, join points are certain well defined points in the execution flow of the process. Since WS-BPEL process consists of a set of activities, each WS-BPEL activity is a possible join point. Pointcuts identify particular join points by filtering out a subset

Message Temporal Constraint Template	
History-Based Pointcut	**Recovery Advice**

E_1: *exit*(//receive[@pt="hotel"@op="cancel"]) {:$outcome/confirm==true:}
E_2: *exit*(//receive[@pt="flight"@op="cancel"]) {:$result/confirm==true:}
E_3: *entry*(//reply[@pt="customer"@op="cancel_Trip"]
Pattern: all(E_1, E_2) ***precede*** E_3
Range: ***around*** //scope [@name = "confirm"]
	{
	⟨partnerlinks⟩
	⟨partnerlink name="agency" partnerLinkType="supervisePLT"…/⟩
	⟨/partnerlinks⟩
	⟨scope⟩⟨sequence⟩
	⟨compensate target = "arrangeTrip"/⟩
	⟨invoke partnerlink = "agency" portType = "supervisePT"
	operation = "notify" inputVariable = "trace"/⟩
	⟨exit/⟩
	⟨/sequence⟩⟨/scope⟩
	}

Fig.2. Stateful aspect based template of Example 1.

of all the join points in the process. The attributes of an activity can be used as predicates to choose relevant join points[13]. Similar to the work in [5, 13], we choose XPath as the pointcut language since WS-BPEL processes are XML-based documents. The set operators XPath provides, such as union operator and intersect operator, can be used to combine pointcuts that select different types of activities.

However, in most aspect extension to WS-BPEL language, the observation that an aspect can make are confined to the current state. "Stateful aspect", the advanced feature in AOSD, is not supported currently. This technique allows an aspect observes the events occurred in the system execution, and when certain events of interest take place, the aspect runs extra code of its own. The ability to express temporal relations between events is a key feature towards the expressive pointcut language. We propose a history-based pointcut for the specification of monitoring requirements based on events occurred during the process execution.

As this pointcut language aims to specify the behavior constraints, more concretely, to specify that the patterns of interest cannot be violated within a range, it consists of two parts: *pattern* and *range*, where *pattern* specifies the constraint on the temporal relation of events and *range* specifies the extent of the process execution over which the pattern holds.

We think that *range* is especially useful for pattern monitoring. It reduces the number of events needed to be checked at runtime. Further, it can control the faults within defined boundaries by triggering suitable fault handling action timely when an erroneous behavior is observed.

Though the proposed pointcut originates from the Property Specification Pattern[14], it shares the same idea with some runtime monitoring languages. For example, many researches[15,16] use WS-Policy[17] to express the monitoring requirements. WS-Policy is a part of the Web Services Policy Framework. This framework also includes WS-PolicyAttachment[18] that specifies how a policy document can be attached to WSDL documents, UDDI entries, and general XML files representing the subject for which the policy holds. The concept of range is similar to the WS-PolicyAttachment. It specifies the extent of the process execution over which the pattern holds.

Fig.3 presents, in a semi-formal way, the syntax of history-based pointcut. Before we give the definition of patterncut, we explain the tracecut firstly.

Tracecut. It specifies the events of interest occurred during the process execution. The events can be primitive or complex, which are captured by *primitive_trace* and *complex_tracecut* respectively. Also, *primitive_tracecut* can be named.

⟨*history_pointcut*⟩ → ***pattern*** ⟨*pattern*⟩ ***range*** ⟨*range*⟩;
⟨*pattern*⟩ → ⟨*quantitative_pattern*⟩|⟨*qualitative_pattern*⟩;
⟨*qualitative_pattern*⟩ → ⟨*tracecut*⟩ ***absent***
| ⟨*tracecut*⟩ ***exist*** [[***at least***] | [***at most***] ⟨*n*⟩ ***times***]
| ⟨*tracecut*⟩ ***precede*** ⟨*tracecut*⟩
| ⟨*tracecut*⟩ ***leadto*** ⟨*tracecut*⟩;
⟨*qualitative_pattern*⟩ → ⟨*tracecut*⟩ ***leadto*** ⟨*tracecut*⟩ ***within*** ⟨*c*⟩ ***time units***
| ⟨*tracecut*⟩ ***precede*** ⟨*tracecut*⟩ ***at least*** ⟨*c*⟩ ***time units***
| ⟨*tracecut*⟩ ***exist*** [[***at least***] | [***at most***]] ⟨*n*⟩ ***times*** ***within*** ⟨*c*⟩ ***time units***
| ⟨*tracecut*⟩ ***exist at least every*** ⟨*c*⟩ ***time units***
⟨*tracecut*⟩ → ⟨*primitive_tracecut*⟩ | ⟨*complex_tracecut*⟩ |⟨*named_tracecut_ref*⟩;
⟨*named_tracecut_ref*⟩ → *id*;
⟨*named_tracecut*⟩ → *id* ::= ⟨*primitive_tracecut*⟩;
⟨*complex_tracecut*⟩ → ⟨*modifier*⟩ (⟨*tracecut*⟩ [, ⟨*tracecut*⟩]*);
⟨*modifier*⟩ → ***seq*** | ***all*** | ***any***;
⟨*range*⟩ → ***global*** | ***before*** ⟨*pointcut*⟩ | ***after*** ⟨*pointcut*⟩ |***around*** ⟨pointcut⟩;
⟨*primitive_tracecut*⟩ → ***entry*** (⟨*pointcut*⟩ [{:*condition*:}]) | ***exit*** (⟨*pointcut*⟩[{:*condition*:}]);

Fig.3. Semi-formal syntax of the history-based pointcut.

Primitive_Tracecut. It defines the lexeme of declarative patterns, capturing individual event of interest in the execution trace. For each activity, two events occur at its activation (entry into the join point) and its completion (exit from the join point). Two kinds of primitive tracecut are provided to capture the activity events. Activation event can be captured through the use of *entry* primitive tracecut, and completion event can be captured through the use of *exit* primitive tracecut.

Both of these take a pointcut as parameter, which can expose the state surrounding the specified join point. Hence, it is possible to specify the condition to the variable of the join point. For example, *exit*(//receive[@partnerlink="hotel" @ operation="cancel"]){:$outcome/confirm == true:} captures the completion events of receive activity that cancellation request from the user has been accepted by the hotel service. Currently, the *condition* supports the typical Boolean operators, such as && (and) and || (or), relational operators, such as ⟨, ⟩ and ==, and mathematical operators such as +, −, ∗ and /.

Complex_Tracecut. Based on primitive tracecut, we define three kinds of complex tracecut operator *any*, *seq* and *all* to capture the complex event. Also, more complex event can be defined based on the combination of these operators.

1) *any* operator takes a set of tracecut as input. The complex event is captured when any event of them occurs.

2) *seq* operator takes a list of n ($n > 1$) tracecuts as its parameter, e.g., $seq(E_1, E_2, \ldots, E_n)$. It specifies an order in which the events of interest should occur sequentially. However, it does not allow an arbitrary number of events specified in $E_1, E_2, \ldots, E_n$ to appear between the two events addressed by two consecutive parameters. The reason why we limit this is considering the filtering semantics in AOP which will be explained in Subsection 4.2.

3) *all* operator takes a list of tracecuts as input. If all specified events occur, the complex event is captured. This operator does not specify the order of events occurred.

Patterncut. It specifies the constraints on the temporal ordering of events, and is classified into qualitative pattern and quantitative pattern. Qualitative pattern specifies abstract temporal relationship of events and quantitative pattern specifies concrete timing-based temporal relationship of events (E, E' represents the event captured through tracecut).

Qualitative Pattern. At present, it includes the following five pattern operators.

1) *absent* operator restricts the specified event does not occur (within the given range).

2) *exist* operator states the specified event must occur.

3) *bounded exist* operator extends *exist* with lower and upper bounds on the number of event occurrence.

4) E *precede* E' states that there must be at least one event E before E' occurs. One may think E enables E'.

5) E *leadto* E' states that E must eventually be followed by E'. In essence, this specifies a cause-effect relationship between E and E'.

Quantitative Pattern. To express timing-based constraints, based on [19], we introduce three time-related limits: *at least every* c *time units*, *within c time units*, *at least c time units*, accompanied by the qualitative patterns. The following combinations are supported for the moment.

1) E *leadto* E' *within c time units* restricts the maximum amount of time that passes after E occurs until E' becomes true.

2) E *precede* E' *at least c time units* restricts the minimum amount of time that E precedes E'.

3) E *bounded existence k times within c time units* restricts the maximum amount of time that passes for "E *bounded existence c times*" holds.

4) E *exist at least every c time units* describes the periodic satisfaction of an event occurrence. It restricts the amount of time in which E has to occur at least once.

Range. At this stage, the range we defined includes *global*, *before*, *after* and *around*.

1) *global* refers to the entire history of the process execution.

2) *before* concerns the initial portion of the history up to first execution of the specified activity (designated by the pointcut).

3) *after* concerns the portion after the completion of the specified activity, which is firstly encountered during the process execution.

4) *around* concerns the portion "during the execution of the specified activity". This range may occur repeatedly over the course of the process execution.

In [11], Filman and Havelund have specified that an expressive event-based pointcut language should express: 1) abstract temporal relationship; 2) abstract temporal quantifiers; 3) concrete temporal relationship referring to clock time, 4) cardinality relationship on the number of times some event has occurred; 5) aggregation relationships for describing sets of events. Our proposed pointcut language satisfies these requirements. For example, *leadto* specifies abstract temporal relationship, *bounded exist* specifies abstract temporal quantifiers, and *complex_tracecut* specifies the aggregation of events.

4.2 Semantics of History-Based Pointcut

So far, we described the semantics of history-based pointcut informally. To enable automated tool support for runtime validation, we need to formally define their semantics. In our approach, we choose FSAs as their underlying semantics since it is easily understandable, mathematically well defined and can be directly used for online trace matching.

Although in [20] the authors have given the FSA semantics of Property Specification Pattern, its skipping based semantics (where any event may be skipped during the match) is generally favored by runtime verification community. Here, we give its FSA notation based on filtering semantics (where every relevant event must be matched), which is shared in the AOP community[21]. An important difference between the two is that under filtering semantics, taking the regular expression "AB" as an example, it means "A implies next B", while under skipping semantics, it means that "A implies eventually B". Fig.4 gives the FSA notation of the pattern operators based on filtering semantics. The semantics of timing-based patterns is described by timed automata[22] (x denotes clock).

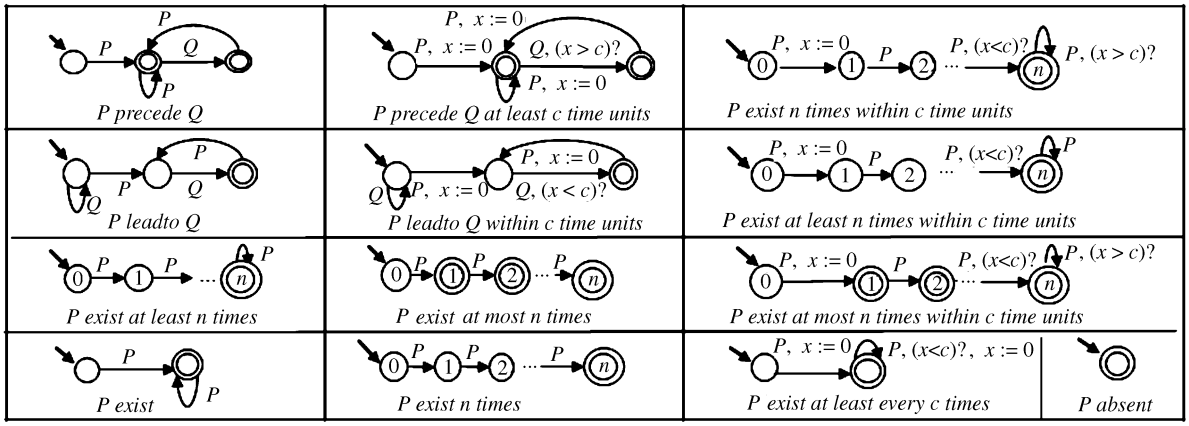

Fig.4. FSA semantics of pattern.

For the complex event operator *any*, *seq* and *all*, as the constructed complex event expressions are equivalent to regular expressions, it is possible to detect complex event using FSA[23]. We do not give their FSA notation here (also the constructed FSA is based on filtering semantics).

4.3 Advice

An advice corresponds to the code that is executed when the specified behavior constraint is violated. We choose WS-BPEL as an advice language for consistency with the original BPEL service implementation[5,13].

Like AO4BPEL[7], the proposed stateful aspect allows to define the variable, partnerlink, etc. In some cases, the advice activity may require context information about the running process. We provide a special variable $\$trace$ to hold information about the current process such as process name, process id, and the already identified event trace in the pattern monitoring.

If specified pattern is violated, some actions need to be performed to manage the running process. For example, Fig.2 shows that if the constraint is violated, the advice first compensates the already completed activity, then notifies the process manager with the violation information, and finally terminates the process execution. However, WS-BPEL only provides some basic and limited fault handling mechanism. To facilitate specification of the recovery advices, we provide several extended actions: *retry*, *alternate* and *skip* to supervise the running process. These actions work on a single process instance. Therefore, the performed action is only valid for the life-span of a single process instance.

• *retry*(S, *number*, *duration*)

retry action repeats execution of the activity S (specified in XPath expression). *number* specifies the upper bound of retry times, while *duration* specifies how long it should be between two executions of the specified activity. For instance, to Example 2, ⟨retry activity="//confirm/scope[@ name='cancel_hotel']" number= "3" duration= "PT5S"/⟩ specifies a retry of "cancel_hotel" activity 3 times with 5 seconds delay between retries.

• *alternate*(S, *url*, *servicename*)

alternate action allows replacing a partner service in activity S by dynamically binding to another functionally equivalent service (identified by *url* and *servicename*). Currently, this action requires the replacement service has the same interface with the substituted service.

• *skip*(S)

This action specifies that activity S need not be executed due to such reasons as the time constraint mentioned in Example 4.

To add a new activity, we just need to add the corresponding WS-BPEL codes of the activity in the advice part of the template. Note that the actions mentioned above can be combined with WS-BPEL constructs to express complex fault handling logic. For example, if the skipped activity and the successor activities have data dependencies, these values can be provided using *assign* activity.

5 Implementation and Tool Support

In the previous section we elaborate the design of AOP-based templates. However, to supervise the business processes at runtime, we need to capture the events occurred during the process execution, check its conformance with predefined behavior constraints and recover the process from the abnormal state in case of violation. In this section, we introduce AOP technology based implementation to support pattern monitoring and advice enforcement. As a proof-of-concept, we implemented the prototype on the top of our developed service composition platform ONCE-BPEL. It consists of aspect

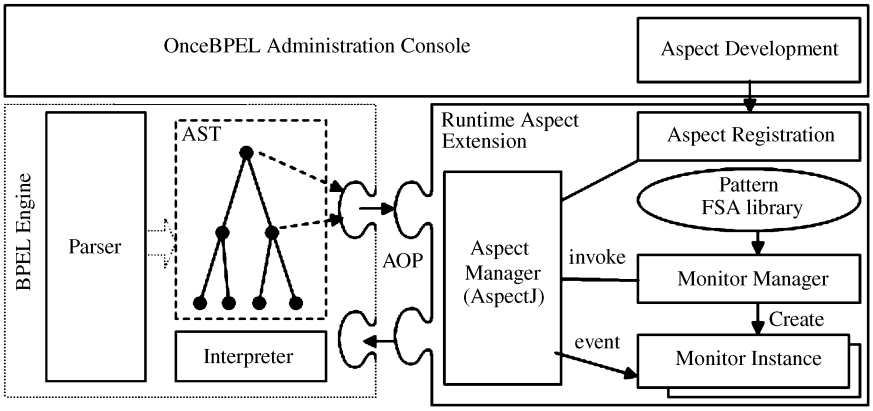

Fig.5. Framework of prototype.

development environment and runtime aspect extension. Fig.5 shows the overall framework of the prototype.

5.1 Development Environment

The development environment assists the developer in instantiating the stateful aspect based templates. To specify the behavior constraint, the users can select the pattern and range operators and provide the associated parameters. Then they can specify the corresponding recovery action.

Once both the behavior constraint and recovery advice are provided, the development environment generates two outputs: the aspect definition file and monitor configuration file.

An example of the aspect definition file is given in Fig.6. It is an XML file that consists of primitive tracecut information (called *eventcut*) and related recovery advice. Two special predefined functions: *activatepattern* and *completepattern* are provided to manage the lifecycle of the pattern and we will discuss them in Subsection 5.2.3.

Monitor configuration file (see Fig.7) saves the information about pattern operator. Used together, they allow the runtime environment to supervise the running processes.

```
⟨aspect processName ="TravelPlan" type="process"⟩
  ⟨partnerlinks⟩
    ⟨partnerlink name ="agency" partnerlinkType=
    "SupervisePLT" myRole="caller"/⟩
  ⟨/partnerlinks⟩
  ⟨variables⟩⟨variable name ="trace" messageType =
    "traceInfoMsg"/⟩⟨/variables⟩
  ⟨eventcut name="E1" designator="exit"⟩
    ⟨pointcut condition="$outcome/confirm==true"⟩
      //receive[@partnerLink='hotel' and @operation=
        'cancel']
    ⟨/pointcut⟩
  ⟨/eventcut⟩ ···
  ⟨pointcutandadvice⟩
    ⟨pointcut⟩//reply[@partnerLink='customer' and @ope-
    ration='cancelTrip']⟨/pointcut⟩
    ⟨advice type = "around"⟩⟨activatepattern/⟩⟨proceed/⟩
    ⟨completepattern/⟩⟨/advice⟩
  ⟨/pointcutandadvice⟩
  ⟨recovery⟩
    ⟨scope⟩⟨sequence⟩
      ⟨compensate target = "arrangeTrip"/⟩
      ⟨invoke partnerlink = "agency" portType = "supervise-
        PT" operation = "notify" inputVariable = "trace"/⟩
      ⟨/exit⟩
    ⟨/sequence⟩⟨/scope⟩
  ⟨/recovery⟩
⟨/aspect⟩
```

Fig.6. Example of aspect definition file.

```
⟨?xml version ="1.0" encoding = "UTF-8"?⟩
⟨monitor name = "monitor1" type = "process"⟩
  ⟨process name = "TravelPlan"/⟩
    ⟨pattern type = "precede"⟩
      ⟨parameter name = "para1" value = "CE"⟩
        ⟨composite type = "all"⟩
          ⟨parameter name = "para1" value = "E1"/⟩
          ⟨parameter name = "para2" value = "E2"/⟩
        ⟨/composite⟩⟨/parameter⟩
      ⟨parameter name = "para2 " value = "E3"/⟩
    ⟨/pattern⟩
⟨/monitor⟩
```

Fig.7. Monitor configuration file.

The development environment also supports process-level and instance-level aspect deployment. With process-level deployment, the deployed aspect will function on every running process instance. With instance-level aspect deployment, only some process instances are affected by the aspect. To achieve this, the

prototype allows users to specify some conditions on the correlation set (e.g., the customerID) in instantiating the templates.

5.2 Runtime Aspect Extension

The runtime aspect extension is the dynamic weaver, which enables monitoring the event trace occurred during the process execution and weaves the advices with the processes when the specified behavior constraint is violated. To support this, our implementation is made up of the following main components. Similar extensions can be considered for other WS-BPEL engine implementation.

5.2.1 Architecture

WS-BPEL Engine. Our WS-BPEL engine is an interpreter and its implementation revolves around the runtime visit and management of an internal tree based representation of WS-BPEL processes. It contains one visit method for each WS-BPEL instruction and traverses, from top to bottom, the ASTs that represent the WS-BPEL documents. These trees are not only strictly typed to meet the pattern requirements but also based on the DOM API to enable XPath selections of nodes. A thorough study led us to define our concerns (using AspectJ) as before and after it visits the activity nodes.

Aspect Registry. It saves all deployed aspect entities information. When the aspect definition file is deployed, the pointcut expression is evaluated. This evaluation operates on AST and returns a set of matching nodes for each pointcut expression. Then the information of the returned activity nodes, along with the associated *eventcut* information (e.g., event name, designator type) is stored into an internal structure of the aspect registry. As such, dynamic aspect addition and removal becomes possible. When an aspect is undeployed at runtime, we just need to delete the corresponding registration information.

Monitor Manager. It maintains all deployed monitors and includes a parser. According to Pattern FSA Library and monitoring configuration file, the FSA of the history based pointcut will be constructed. As we did not construct a single automaton, an event tree is constructed to organize the relationship of multiple automata (as shown in Fig.8).

Aspect Manger. Aspect manager represents the main advice which is weaved into the execution environment. After weaving, this component has direct access to the internal representation of the process and its state. This component is responsible for managing all the steps in the pattern monitoring.

Before or after an activity is executed, aspect manager is inserted that defers execution to the aspect runtime infrastructure. It looks up all the *eventcut* information registered for that particular activity based on the current execution context. This infrastructure will trigger any aspects that apply to the join point or, when no aspects are applicable, return to the normal execution. The aspect manager also maintains the relation between the process instance and the monitor instance, and dispatches event to the right monitor instance if the specified eventcut condition is satisfied under the current process execution context. We will defer a more in depth analysis of pattern validation process in Subsection 5.2.3.

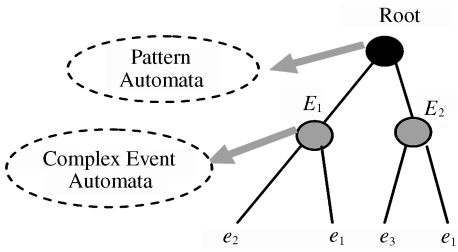

Fig.8. Event tree structure.

To support dynamic weaving, AO4BPEL[24] implemented an aspect-aware engine. It modifies the activity lifecycle of the WS-BPEL engine with the dynamic checks for aspects. The aspect runtime builds a wrapper around the WS-BPEL interpreter using object-oriented technology by applying subclasses to add additional functionality. During the process execution, it checks whether there is an aspect with a pointcut that matches the current activity. If such an aspect is found, the respective advice activity will be executed according to advice type.

Different from their proposed mechanism, our implementation utilizes the visitor design pattern: the WS-BPEL engine is implemented as visitor of an abstract syntax tree representing a WS-BPEL process. Interestingly, the runtime aspect extension has itself been implemented as an aspect weaved with WS-BPEL engine using AspectJ, which keeps the base system (i.e., WS-BPEL engine) and the extension as separate as possible. Thus it supports the aspect extension component to be plugged in or unplugged from the WS-BPEL engine easily.

5.2.2 Optimization

The runtime weaving technology mentioned above was essentially trap-based. The proposed aspect runtime extension contains a pre-processor that registers eventcut information at all possible join points before runtime. The main problem of this pre-processing approach is performance. This overhead stems in part from the naïve interception system, namely inserting

```
Retry(S, n, duration) {
    ⟨variable name =“n” type = “xsd:int”/⟩
    ⟨while condition = “n > 0”)
      ⟨scope name = “S”⟩
        ⟨faulthandler ···⟩⟨catch⟩
          ⟨switch⟩
            ⟨case condition = “n = 0”⟩
              ⟨rethrow/⟩⟨/case⟩
            ⟨otherwise⟩
              ⟨wait for = “$ duration”/⟩
                ⟨assign⟩ n := n − 1⟨/assign⟩
            ⟨/otherwise⟩⟨/switch⟩⟨/catch⟩⟨/faulthandler⟩
        ···
      ⟨assign⟩ n := 0 ⟨/assign⟩
    ⟨/scope⟩⟨/while⟩
}
```

```
Alternate(S, url, servicename) {
    ⟨sequence⟩
        ⟨assign⟩
            ⟨copy⟩
              ⟨from⟩⟨wsa:EndpointReference⟩
                ⟨wsa:Address⟩url⟨/wsa:Address⟩
                ⟨wsa:ServiceName⟩servicename
                ⟨/wsa:ServiceName⟩
              ⟨/wsa:EndpointReference⟩
            ⟨/from⟩
          ⟨to partnerLink = $S.partnerlink/⟩
        ⟨/copy⟩
    ⟨/assign⟩
    S;
  ⟨/sequence⟩
}
```

Fig.9. WS-BPEL codes of *retry* and *alternate* actions.

traps at all possible join points. To reduce the overhead incurred, for the instance-level aspect deployment, some optimization technique is adopted. We make the best use of the *range* part in the proposed history-based pointcut. That is, the possible join point traps are annotated only when the range activity is entering. When the range is exited, the traps are all un-deployed from the aspect registry. For example, to the range *after*, the pattern related join points will be deployed right after the specified range activity completes. To the range *before*, the aspect information is undeployed from the aspect registry just before the specified range activity activates.

To the range *around*, as the XPath based pointcut can express the nested relationship of join points, it is possible to merge the range-related pointcut with the pattern-related pointcut during the deployment of aspect definition file. Then we can make annotations to the possible join points according to the merged pointcut, thus it greatly reduce the events to be generated. This kind of optimization can be applied to both instance-level and process-level aspect deployment.

5.2.3 Behavior Constraint Validation Process

As we construct pattern automata and complex event automata separately, to detect the violation to pattern constraint, an event tree is constructed to organize the relationship of multiple automata (see Fig.8). The event tree consists of a root node, non-terminal nodes (N-node), terminal nodes (T-node) and edges. Root node represents pattern automaton, N-node represents complex event automata and may have several incoming and several outgoing edges. T-node represents primitive event and possibly has several outgoing edges. When a primitive event occurs, it activates the terminal node that represents the event. This in turn activates all nodes attached to it via outgoing edges. When N-node receives an event, the corresponding automata will match the incoming data and if the specified complex event happens, it sends the corresponding symbol to the parent nodes. Both control and data flow from the leaf nodes to the root node in the event tree. A finite sequence of events is considered valid if taking the events sequentially leads the pattern automata into a final state. A violation will occur if the pattern automata receive an event that is unspecified or the clock condition is not satisfied at the current state. This will lead the automata into a violation state (not shown in Fig.4 for brevity).

The range operator defines the boundaries for the pattern monitoring, which can reduce the events to be matched at runtime. To implement this, we utilize the advice type “before”, “after” and “around” introduced in AOP, and define two internal functions: *activatepattern* and *completepattern* to manage the lifecycle of the pattern. For each monitor instance we use an internal *flag* to indicate whether the specified pattern is being identified. Function *activatepattern* initializes the monitor instance, and sets the *flag* true. The constructed automata (including pattern automata and complex event automata) could start afterward. Function *completepattern* checks whether the pattern automata is in a non-terminal state or a violation state. In these two cases, the corresponding recovery advices will be weaved into the engine by the aspect manager to recover the process from the abnormal state. This

function also sets the *flag* false, which will stop matching the event afterward. Note that to the range *before* and *global*, function *activatepattern* will be invoked immediately after the monitored process starts. To the range *after* and *global*, if the pattern automaton enters into the violation state, the corresponding recovery advices will be executed immediately.

5.2.4 Advice Enforcement

If the violation is detected, the recovery advice is loaded and executed according to the definition.

To enforce *retry* and *alternate* actions, which may access the internal variable and change the behavior of the running process, we transform them into the WS-BPEL codes (see Fig.9). These WS-BPEL codes are then weaved into the engine to manage the running process. Note that, the implementation of *alternate* action utilizes the dynamic partner service resolution WS-BPEL language provides.

To support *skip* action, an ordinary aspect is generated by the aspect manager at runtime, where the pointcut identifies the activity to be skipped, the attribute *condition* specifies the impacted process instance and the type of the advice is *around* (as shown in Fig.10).

```
Skip(S){
  ⟨aspect name = "skip" condition = "···"⟩
    ⟨pointcut⟩
      //xpath to identify activity S
    ⟨/pointcut⟩
    ⟨advice type = "around"/⟩
  ⟨/aspect⟩
}
```

Fig.10. Aspect codes of *skip* action.

6 Experiments and Evaluation

To estimate the overhead incurred by the proposed pattern monitoring approach, the performance impact on the monitored processes, we conducted several experiments. Note that here we focus on the matching of the event pattern, and our measurements do not include the execution time of the recovery advice.

In the experiments, we used a process with 150 activities, which consist of sequences of nested *while* constructs, with atomic activities that each invokes a Java class and some run in parallel within flow activity. The tests were performed on P4 2.8GHZ, 1GB RAM memory, running Windows 2003. The WS-BPEL engine (in the experiment we use our developed ONCE-BPEL engine) ran on Tomcat 5.5.

Our experimental setup consists of two configurations: one without runtime aspect extension and one with runtime aspect extension. We measured execution time of the process from the perspective of the client using the Round Trip Time (in seconds), which is a measurement from the time the client sends a request to the point when it receives the fully reply from the process.

In the first experiment, the deployed aspects involve about 30% of WS-BPEL events, the deployment type is the process-level and range is *global*. We compare the performance overhead incurred by aspect monitoring.

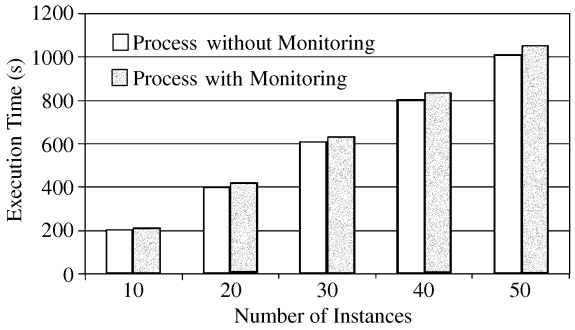

Fig.11. Performance overhead.

Fig.11 shows, for a varying number of process instances, two measurements of total execution time for an entire workload. The first column in each set shows the execution time of the process instances without monitoring. The second column shows the execution time of the processes with monitoring. Clearly, the overhead due to monitoring is very low. The average time of the process execution is only 5% higher than (the first column) when monitored. Obviously, all the results are affected by the scalability of the WS-BPEL server itself. We can see that the execution time grows linearly with the number of concurrent processes.

In the second experiment, we measured the performance overhead when monitoring different percentage of the WS-BPEL events. We ran the experiment with 50 process instances. The aspects with the process-level deployment involve 10% to 80% of WS-BPEL events. Fig.12 shows the execution time grows moderately with the percentage of the monitored events. In practice, as typical business process is large with only small parts being relevant for a particular monitoring task, the right column will be close to the left column.

In the following experiments we concentrate on the effect of applying optimization technique to reduce the performance overhead.

In the third experiment, we compared the performance overhead after applying optimization technique

to the range *around*. The deployed aspects involve 60% WS-BPEL events, and the deployment type is the process-level.

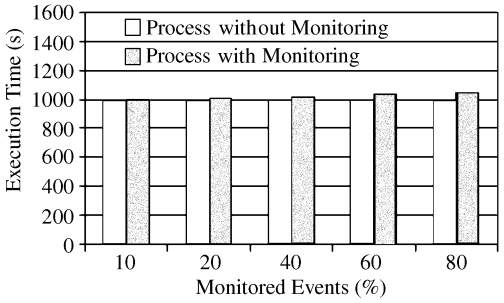

Fig.12. Varying the number of WS-BPEL events.

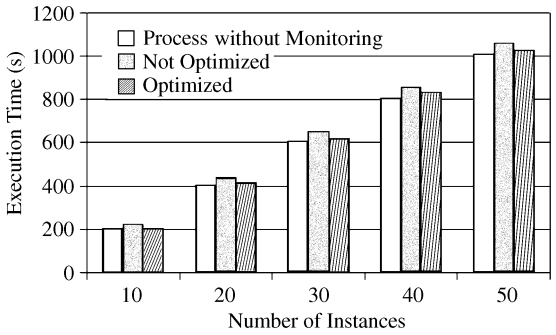

Fig.13. Impact of optimization to range *around*.

Fig.13 shows three measurements of total execution time, varying the number of process instances. The first column in each set shows the execution time of processes without monitoring. The second column shows the execution time of the processes when monitored but with no optimization. The last column shows the execution time with optimization. From this figure, we can see that before optimization, to each monitored process, the average performance overhead increases 10%, and after optimization, decreases about 5%.

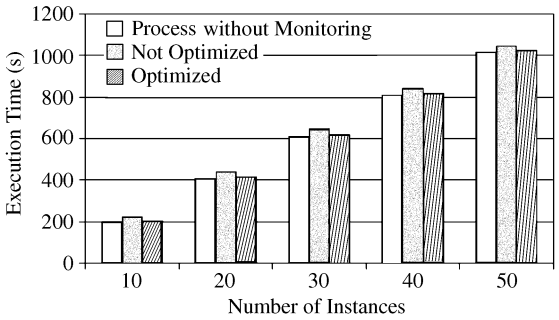

Fig.14. Impact of Optimization to range *after*.

In the last experiment, we consider the effect after applying the optimization to the range *after*. The deployed aspect involves 80% WS-BPEL events. The deployed type is instance-level and the amount of monitored processes is 15% of the running processes. We have similar results as the third experiment. Fig.14 shows that before optimization, the average performance overhead increases 11%, and a performance gain of 7% is achieved after optimization to the range *after*.

Of course, in general the performance improvement is related with the characteristics of the monitored process structure and the specified behavior constraints.

7 Discussion

In this section, we consider some issues surrounding the concept of history-based pointcut.

Join Point and Event. One may think that we could directly use the join point to represent the events occurring in the execution trace. However, the activity-based join point model does not easily support the expression of event patterns because its join points are not well-ordered.

To see this, consider the example OTA process in Fig.2. A straightforward interpretation of this situation might be that activity scope *arrangeTrip* executes before activity invoke *flight*. However, consider applying *before* and *after* advice to the execution of each of these activities. The *before* advice on scope *arrangeTrip* would execute before that on invoke *flight*, but the *after* advice would execute in the reversed order. In other words, the execution pointcut does not describe a discrete event but an interval from its start of the activation to its end from a trace-based perspective. This problem becomes more obvious when the sub-process extension for WS-BPEL[25] is considered.

Extensibility. Runtime monitoring is one of the typical crosscutting concerns in developing service-based system. To solve this problem, we proposed a history-based pointcut to specify the behavior constraint to the WS-BPEL processes, including qualitative constraint pattern and quantitative constraint pattern. Note that the proposed pointcut is extensible. For example, the *complex_tracecut* part of the proposed history-based pointcut can be extended to specify regular expression based patterns: $*$ specifies the event can occur zero or more times, $+$ specifies the event can occur one or more times, etc. In [4], the authors define five ranges. At this stage, besides the *global* range, we support three activity-related ranges, such as *before*, *after* and *around*. The reason we do this is considering the nested structure of WS-BPEL processes. We think these four ranges are sufficient to express most behavior constraints now. More pattern and range operators can be added easily if we find it necessary later.

Limitation. Though the complex monitoring requirements could be specified using traditional aspects by keeping an explicit state variable in the aspect code, such a manual implementation is an error-prone task. Our solution provides linguistic support for the stateful aspect, which frees the developer from the burden of writing the state bookkeeping code and allows for a clean implementation of stateful aspects.

At the moment, the monitoring requirements need to be manually specified by the process developer. It would be better to generate them automatically from some high-level specification or policy. For example, some behavior constraints to the partner services can be generated from the policy of the partner service[17] or the service level agreement between the partner and process (i.e., WS-Agreement[26]), and constraints to the process itself can come from some high-level software requirements specified in UML[27], etc. We will investigate these problems in the future work.

8 Related Work

In this section, we first discuss the existing aspect extensions to WS-BPEL language. Then some recent research efforts on runtime monitoring of WS-BPEL processes are presented.

8.1 Aspect-Oriented Extension to WS-BPEL

AO4BPEL[11] is an aspect-oriented extension to WS-BPEL that allows for more modular and dynamically adaptable Web service compositions. Each WS-BPEL activity is a potential join point, and the XPath is chosen as the pointcut language. To support middleware concerns such as security, reliable message and transaction, two special internal message join point designators *messagein*/*messageout* are introduced. The aspect runtime is the dynamic weaver, which modifies the interpretation flow of the activity lifecycle by checking if any aspect matches the current join point activity. AO4BPEL also supports to separate the business rules from the business process[28], but it does not consider the complex business rules, whose triggering events relies on the execution history of the core application.

Courbis and Finkelstein[5] present an aspect-oriented extension to WS-BPEL, which is very similar to AO4BPEL. They also use XPath as a pointcut language, and the advice language can be Java or WS-BPEL. They focus on the dynamic adaptation, but do not address the issues of crosscutting concerns in Web service composition and how to modularize them using aspects. The dynamic weaving mechanism modifies the abstract syntax tree representation of the WS-BPEL process to annotate some aspect information on it. In order to weave aspects, the engine is suspended at some points and the AST tree is transformed, e.g., by adding or removing activities from/to the process, which has serious implications on performance. Unlike this approach, the advice in our approach only acts on the impacted process instance.

As far as we know, there is only one proposal Padus[12] that supports stateful aspect extension to WS-BPEL language. Different from the above mentioned work, the basic pointcut is based on a logic-based programming language, and pointcut matching is done by Prolog engine. To support stateful aspect, their work specifies pattern with regular expression, which is not enough to express complex behavior constraints. The implementation of Padus uses static weaving at source code level. The logic which traces the process history is added by injecting some pieces of advice into the process, which performe some book keeping on the state of the process by employing process variables to store the state information. The shortcoming of this approach is that it has some degree of intrusiveness. After weaving, the process will be blended with a lot of *assign* and *switch* activities that have nothing to do with the business logic. Also, static weaving is not flexible, as it does not support to modify/add/remove aspects during the process execution.

8.2 Runtime Monitoring

To ensure the correctness of a distributed process, some static techniques can be used to verify the behavior of Web services, such as the work in [29, 30]. While static analysis for checking Web services composition against properties is appealing, they have some limitations: since Web services communicate via infinite-length channels, the problem is decidable only under certain condition. Further, some advanced features in WS-BPEL, for example, the endpoint reference (to dynamically determine the partner to talk to), are not considered. Also, existing techniques are unable to deal with complex message interaction[31]. Instead, we choose dynamic analysis via runtime monitoring.

At this stage, a lot of work has been studied in the area of runtime monitoring WS-BPEL processes, which can be classified into two kinds, assertion-based monitoring and event-based monitoring.

In [32], Baresi *et al.* propose an assertion-based approach to monitoring the WS-BPEL processes, where the monitors are specified as assertions that annotate the WS-BPEL codes, which are then automatically translated to "monitored processes". In [33], they further propose a design process model for the definition of supervised processes and an AOP-based prototype

implementation is also presented.

For the event-based monitoring, Mahbub *et al.*[3] developed a framework for monitoring requirements of WS-BPEL processes. Their approach uses event calculus (EC) to specify the requirements to be monitored. During the execution, all process events are generated and transformed to terms of events in EC. Run-time checking is interpreted as integrity constraint checking in a temporal deductive database. This approach is less intrusive that monitor proceeds in parallel with the execution of the business process and has a lesser impact on performance. It also leads to a lesser responsiveness in discovering erroneous situations. The monitoring approach we proposed only detects the events of interest and can supervise the running process in case of violation.

Other researches address the reliability problem of WS-BPEL processes[34−36]. In [34], the authors present an approach where a designer defines a WS-BPEL process annotated with some information about recovery actions and then a pre-processing phase, then starting from the "annotated" WS-BPEL, generates a "standard" WS-BPEL, which is understandable for any standard WS-BPEL engine. Inspired by this work, [35] proposes a declarative approach to enhancing the reliability of WS-BPEL processes. ECA rules based fault handling logic, such as *retry*, *alternate* and *skip*, are integrated with business logic through static weaving before deployment. In [36], a set of extensible recovery policies is proposed to specify how to recover from typical faults in Web services composition. The enforcement of the recovery policies is delegated to the underlying messaging middleware. Different from their work, we focus on pattern monitoring and the advices are dynamically weaved at the process level.

9 Conclusion and Future Work

Web services compositions are rapidly emerging as critical building blocks for process integration. Monitoring the execution of such composition is important to enhance the reliability of the process.

In this paper, we have discussed limitations of existing aspect extension to WS-BPEL language. We introduced stateful aspect based template, presented and implemented an aspect-oriented framework as a solution to provide monitoring and recovery support for the composite Web services.

Recently there have been several proposals for enhancing the AspectJ with the ability of trace matching by specifying patterns with the regular expression[37] or context-free grammar[9]. Our work is focused on WS-BPEL and complex event is allowed in the template. An event tree is also constructed to organize the relationship of multiple automata for pattern validation.

This proposal could be improved in a number of ways. First, as discussed in Section 7, currently the stateful aspect template needs to be manually specified by the process developer. It would be helpful if the stateful aspect template could be generated automatically from some high level specifications, such as WS-Agreement, WS-Policy, etc. We will investigate this issue in the future. Second, the recovery actions supported are limited now. In the future work, we plan to explore more complex recovery action to enhance the reliability of WS-BPEL processes.

References

[1] Web services business process execution language (WS-BPEL) v2.0. OASIS. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf.

[2] Delgado N, Gates A Q, Roach S. A taxonomy and catalog of runtime software fault monitoring tools. *IEEE Trans. Software Engineering*, 2004, 30(12): 859–872.

[3] Mahbub K, Spanoudarkis G. A framework for requirements monitoring of service based systems. In *Proc. the 2nd Int. Conf. Service-Oriented Computing*, New York, USA, ACM, Nov. 15–18, 2004, pp.84–93.

[4] Autonomic Computing Initiative. AutonomicComputing. IBM, http://www-03.ibm.com/servers/autonomic.

[5] Courbis C, Finkelstein A. Towards aspect weaving applications. In *Proc. the 27th Int. Conf. Software Engineering*, St. Louis, USA, May 15–21, 2005, pp.69–77.

[6] Avgustinov P, Bodden E, Hajiyev E. Aspects for Trace Monitoring. Formal Approaches to Software Testing and Runtime Verification, Springer, 2006, pp.20–39.

[7] Charfi A, Mezini M. AO4BPEL: An aspect-oriented extension to BPEL. *Journal of World Wide Web*, Springer, 2007, 10(3): 309–344.

[8] Braem M, Verlaenen K, Joncheere N *et al.* Isolating process-level concerns using Padus. In *Proc. the 4th Int. Conf. Business Process Management*, Vienna, Austria, Springer, Sept. 5–7, 2006, pp.113–128.

[9] Allan C, Avgustinov P *et al.* Adding trace matching with free variables to AspectJ. In *Proc. the 20th ACM SIGPLAN Conf. Object Oriented Programming, Systems, Languages, and Applications*, San Diego, USA, ACM, Oct. 16–20, 2005, pp.345–364.

[10] Vand W, Suvee D, Cibran M A. Stateful aspects in JAsCo. In *Proc. the 4th Int. Workshop Software Composition*, Edinburgh, UK, Springer, April 9, 2005, pp 167-181.

[11] Filman R E, Havelund K. Source-code instrumentation and Quantification of Events. In *Workshop on Foundations of Aspect-Oriented Languages. at AOSD*, Twente, Netherlands, ACM, 2002, pp.45–49.

[12] Braem M, Gheysels D. History-based aspect weaving for WS-BPEL using Padus. In *Proc. the 5th Europe Conference on Web Services*, Halle, Germany, IEEE Computer Society, Nov. 26–28, 2007, pp.159–167.

[13] Charfi A, Mezini M. Aspect-oriented Web services composition with AO4BPEL. In *Proc. Europe Conf. Web Services*. Erfurt, Germany, Springer, Sept. 27–30, 2004, pp.168–182.

[14] Dwyer M B, Avrunin G S, Corbtt J C. Patterns in property specifications for finite-state verification. In *Proc. the 21st*

Int. Conf. Software Engineering, Los Angeles, USA, ACM, May 16–22, 1999, pp.411–420.

[15] Baresi L, Guinea S, Plebani P. WS-policy for service monitoring. In *Proc. the 6th Int. Workshop of Technologies for E-Services*, Trondheim, Norway, Springer, Sept. 2–3, 2005, pp.72–83.

[16] Tosic V, Erradi A, Maheshwari P. WS-Policy4MASC — A WS-policy extension used in the MASC Middleware. In *Proc. IEEE Int. Conf. Services Computing*, Salt Lake City, USA, IEEE Computer Society, July 9–13, 2007, pp.458–465.

[17] Bajaj S, Box Don *et al*. Web services policy framework. 2006, http://www-128.ibm.com/developerworks/library/specification/ws-polfram/.

[18] Sharp C (ed.). Web Services Policy 1.2-Attachment (WS-PolicyAttachment), 2006, http://www.w3.org/Submission/WS-PolicyAttachment/.

[19] Konrad S, Cheng B H C. Real-time specification patterns. In *Proc. the 27th Int. Conf. Software Engineering*, Louis, USA, ACM, May 15–21, 2005, pp.372–381.

[20] Simth R L, Avrunin G S, Clarke L A, Osterweil L J. PROPEL: An approach supporting property elucidation. In *Proc. the 22nd Int. Conf. Software Engineering*, Orlando, USA, ACM, June 4–11, 2000, pp.11–21.

[21] Avgustinov P, Moor O D, Tibble J. On the semantics of matching trace monitoring patterns. In *Proc. the 7th Int. Workshop on Runtime Verification*, Vancouver, Canada, Springer, March 13, 2007, pp.45–46.

[22] Alur R, Dill D. A theory of timed automata. *Theoretical Computer Science*, 1994, 126(2): 183–235.

[23] Gehani N H, Jagadish H V, Shmueli O. Composite event specification in active databases: Model & implementation. In *Proc. the 18th Int. Conf. Very Large Data Bases*, Vancouver, Canada, Aug. 23–27, 1992, pp.327–338.

[24] Charfi Anis. Aspect-oriented workflow language: AO4BPEL and application [Ph.D. Dissertation], Damstadt University of Technology, Germany, 2007.

[25] Kloppmann M, Koenig D. WS-BPEL extension for subprocesses. A Joint White Paper by IBM and SAP, September 2005.

[26] Andrieux A, Czajkowsk K, Dan A, Keahey K *et al*. Web services agreement specification (WS-Agreement), 2007, http://www.ogf.org/documents/GFD.107.pdf.

[27] Autili M, Inverardi P, Pelliccione P. A scenario based notation for specifying temporal properties. In *Proc. Int. Workshop on Scenarios and State Machines: Models, Algorithms, and Tools*, Shanghai, China, ACM, May 20–28, 2006, pp.21–28.

[28] Anis C, Mira M. Hybird Web services composition: Business processes meet business rules. In *Proc. the 2nd Int. Conf. Services Oriented Computing*, New York, USA, ACM, Nov. 15–18, 2004, pp.30–38.

[29] Fu X, Bultan T, Su J W. Analysis of interacting BPEL Web services. In *Proc. the 13th Int. World Wide Web Conference*, New York, USA, May 19–21, 2004, pp.621–630.

[30] Foster H, Uchitel S, Magee J, Kramer J. Model based verification of Web service compositions. In *Proc. the 18th IEEE Int. Conf. Automated Software Engineering*, Montreal, Canada, IEEE Computer Society, Oct. 6–10, 2003, pp.152–163.

[31] Web services description language (WSDL 2.0). W3C. http://www.w3.org/TR/wsdl20.

[32] Baresi L, Guinea S. Towards dynamic monitoring of WS-BPEL processes. In *Proc. the 3rd Int. Conf. Service-Oriented Computing*, Amsterdam, Netherlands, Springer, Dec. 12–15, 2005, pp.269–282.

[33] Baresi L, Guinea S, Plebani P. Policies and aspects for the supervision of BPEL processes. In *Proc. the 19th Int. Conf. Advanced Information Systems Engineering*, Trondheim, Norway, Springer, June 11–15, 2007, pp.340–354.

[34] Modafferi S, Conforti E. Methods for enabling recovery actions in WS-BPEL. In *Proc. the Move to Meaningful Internet Systems*, Montpellier, France, Springer, Oct. 29–Nov. 3, 2006, pp.219–236.

[35] Liu H, Li Q, Xiao M J. A declarative approach to enhancing the reliability of BPEL processes. In *Proc. IEEE Int. Conf. Web Services*, Salt Lake City, USA, IEEE Computer Society, July 9–13, 2007, pp.272–279.

[36] Erradi A, Maheshwari P, Tosic V. Recovery policies for enhancing Web services reliability. In *Proc. IEEE Int. Conf. Web Services*, Chicago, USA, IEEE Computer Society, Sept. 18–22, 2006, pp.189–196.

[37] Walker R J, Viggers K. Implementing protocols via declarative event patterns. In *Proc. the 12th ACM SIGSOFT Int. Symposium on Foundations of Software Engineering*, Newport Beach, USA, ACM, Oct. 31–Nov. 5, 2004, pp.159–169.

Tao Huang received his M.S. and Ph.D. degrees from Department of Computer Science, University of Science and Technology of China in 1991 and 1994 respectively. Currently he is a professor and Ph.D. supervisor in the Institute of Software, Chinese Academy of Sciences, and a CCF member. He has published more than 40 research papers and co-authored two books, and undertaken many research projects. His research interests include service oriented computing, middleware, software component technology, distributed object technology.

Guo-Quan Wu currently is a Ph.D. candidate in the Department of Computer Science and Technology, University of Science and Technology of China. He is also spensored by the joint Ph.D. program with Institute of Software, Chinese Academy of Sciences. His research interests include service oriented computing, business process management and aspect oriented software development.

Jun Wei is currently a professor in the Institute of Software, Chinese Academy of Sciences, Beijing, China, also a CCF member. His research interests include service oriented computing, middleware, mobile computing, software engineering. He received his B.Sc. and Ph.D. degrees in computer science from Wuhan University, Hubei, China in 1992 and 1997, respectively. He was a postdoctoral researcher at the Hong Kong University of Science and Technology, China. He has published over 50 papers in international journals and conferences.