Research Paper

Lightweight formal analysis of Web service flows

Shin NAKAJIMA
National Institute of Informatics

ABSTRACT
BPEL (Business Process Execution Language) is proposed as a standard language to describe Web service flows. A flow may contain multiple activities that are executed concurrently, and thus removing faults such as deadlocks or violations of application-specific properties is not easy. This paper proposes techniques to extract a behavioral specification from the BPEL program and to verity it with the model checking technique.

KEYWORDS
Web service, BPEL, model checking, lightweight formal methods, EFA, SPIN

1 Introduction

Service-oriented computing [31] is an emerging software technology for business networks using Web services.[8] The concept of service-oriented computing is that business partners, each acting as a service provider, collaborate with each other for their customers'benefit. In addition, while a provider may offer distinctive services, it sometimes needs to combine more than one Web services into a new service.

Since the collaboration is essential for creating an easy-to-use service, a robust framework is needed to compose lots of specialized services. The essential element is a business process description language to express how Web service providers collaborate together. Since each Web service is implemented by a service provider, which is a self-contained software system having its own threads of control, the business process or flow is essentially a description of collaborations of distributed autonomous computing entities.

Writing correct flow descriptions, however, is not an easy task because the Web flow description is basically a distributed collaboration. While faulty flow descriptions are undesirable from the viewpoint of the conventional software engineering, the situation with the Web service flows is even worse. That is, a faulty description executed in the Internet environment consumes publicly shared network resources. Verifying the Web service flow prior to its execution is essential.[23][24] Actually, there have been various studies on the analysis of the behavioral specifications.[4][9][10][11][16][25][28][29]

This paper proposes to use model checking techniques[6] for analyzing Web service flow descriptions. It focuses on the BPEL (Business Process Execution Language) [7] as a representative language to describe the Web service flows, and uses the SPIN model checker [13] as the verification engine. The concern here is to identify faulty behaviors as early as possible in the development process and not to show the absence of such bugs. The techniques compose a lightweight analysis method as in,[14] in that they constitute a partial but automatic analysis technique rather than a complete but costly analysis method such as using an interactive theorem prover.

First, the behavioral specification is extracted from the BPEL application program in order to represent it in a variant of an EFA (Extended Finite-state Automaton). After that, the EFA model is translated into a Promela source program that is automatically analyzed by using the SPIN model checker. The proposed method employs an abstraction of variables that affect the control aspects of the behavior and provides an adequate support for DPE (Dead-Path Elimination).

Received April 19, 2005; Revised July 12, 2005; Accepted July 13, 2005.
Also with The Grauduate University of Advanced Studies and SORST, Japan Science and Technology Agency
nkjm@nii.ac.jp
DOI : 10.2201/NiiPi.2005.2.5

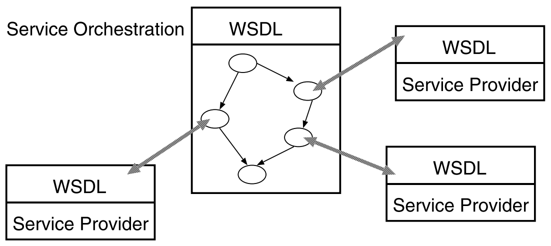


Fig. 1 Web service flows.

These techniques are essential to analyze all the four example cases in the BPEL standard document.[7]

The rest of the paper is organized as follows. Section 2 gives an overview of the Web service architecture and BPEL as a standardized language for describing Web service flows. Section 3 considers the problem of behavioral analysis of BPEL applications. Section 4 introduces the EFA used as the intermediate representation for the behavioral specification of the BPEL applications in order to describe the verification problem for the subject matter. Section 5 presents the method of conducting the behavioral analysis with the SPIN model checker. Section 6 discusses the proposed techniques and compares them with related work. Section 7 concludes the paper.

2 Web service flows

2.1 Web service architecture

A Web service is a building block in the sense that multiple existing services can be used to compose a new service. Fig. 1 is an illustration of the Web service composition.

Each Web service is given an interface description in terms of WSDL (Web Service Description Language).[5] WSDL is a format to describe external interfaces of Web services and it uses the XML technology as its concrete representation. It is essentially an XML document to represent the endpoint interface specification of the service; that is, it contains the necessary information for customers to receive the service. Further, it sometimes needs to combine more than one Web services into a new composite service because a provider may offer a particular service only. And thus a robust framework to realize such composition, a description of *Service Orchestration* in Fig. 1, is desirable.

A typical example of such a *composed* service is a travel agency. A travel agency has a variety of business partners such as airline carriers, hotels, rent-a-car agencies, and railroad companies. The agency receives requests from its customers to make their itineraries with all the necessary reservations of hotels, flights etc. When regarding each of its business partner as an independent Web service provider that has a definite WSDL for the accesses from the outside, the travel agency actually implements a Web service flow. The agency thus collaborates with its partner Web service providers to offer a *composed* service to its customers.

Since each service provider is a self-contained autonomous software system, the composition needs an explicit notion of both control and data flow between Web services that faithfully reflects the causal structure of the component services. Although the service composition should be done in a flexible manner, it is not an easy task to construct correct flow descriptions. This is because service composition is basically a distributed collaboration of many autonomous service providers working concurrently. The flow descriptions may thus show faulty behaviors such as deadlocks and violations of application requirements. Such buggy flow descriptions are obviously undesirable, so verifying the Web service flow description prior to its execution becomes essential.[23][24]

2.2 BPEL

BPEL4WS (Business Process Execution Language for Web Service), or BPEL for short, was proposed as a standard language for describing Web service flows that composed of more than one Web services. BPEL v1.0 was released in July 2002 as a language to supersede both WSFL [19] and XLANG.[32] BPEL v1.1,[7] considered in this paper, is currently being standardized at OASIS as WS-BPEL.

BPEL is a behavioral extension of WSDL (Web Service Description Language).[5] WSDL is basically an interface description language for Web service providers, which describes information that their clients are allowed to access. The client invokes a Web service provider by using WSDL. The invocation is *one-shot*, which means that WSDL does not describe the global states of the providers.

On the other hand, BPEL is a language for expressing behavioral compositions of Web service providers. It can express *a causal relationship* between multiple invocations by means of control and data flow links. BPEL employs a distributed concurrent computation model with variables.

Fig. 2 illustrates some of the primary language elements in BPEL. The main construct of the Web service flow is `Process`, which is a net-based concurrent description connecting activities with control links. Some of the primitive activities, in turn, include sending or receiving messages to or from external Web service providers. Each Web service provider can be seen as a `Port` instance of a particular `Port Type`, which has appropriate WSDL descriptions as its sub-elements. A `Partner Link` specifies which activity is

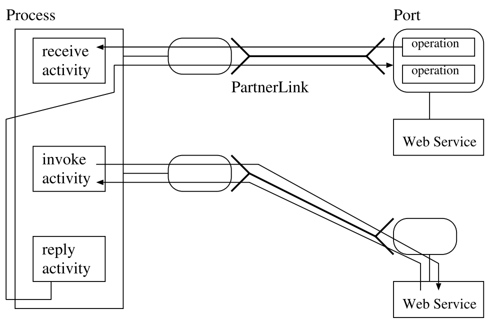


Fig. 2 Primary entities in BPEL.

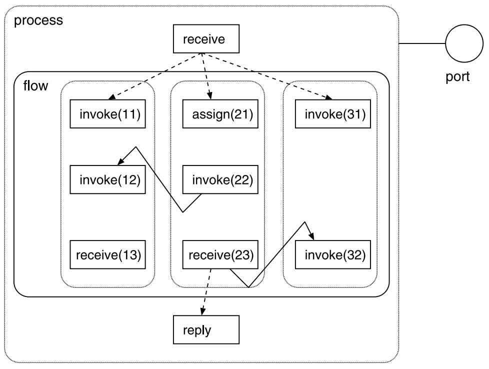


Fig. 3 Purchase order.

linked to a particular Web service provider of the `Port`.

Fig. 3 is an example of a BPEL description that illustrates a `Process` internal. The example indicates that BPEL represents concurrency by means of a `flow` activity in addition to the sequential executions of basic activities. It first waits for an invocation request from the outside with the `receive` activity, and then initiates three concurrent threads through the `flow`. After the concurrent executions terminate, the control goes to the `reply` activity which returns a value to the original outside initiator as the result of the computation process. The schematic diagram also has solid lines (`links`) to indicate control dependencies between some of the basic activities, and they cross the boundary of the concurrent activities.

A program in BPEL actually consists of two sections: `<definitions>` and `<process>`. Example descriptions are found in Fig. 4, which is a fragment of the BPEL program of the example in Fig. 3.

As its full name suggests, BPEL is characterized as a business process description language in the Web service architecture, and it shares many features with work-flow schema languages,[36][37] because the concurrent aspect of BPEL (`flow` activity) is inherited from WSFL.[19] WSFL, in turn, borrows its core ideas from PM-Graph,[17][18] which is a model of workflow systems.

2.3 Formal analysis of behavior

This paper focuses on modeling and analysis of behavioral specifications of BPEL programs. In general, a behavioral specification focuses on the control aspects of the system, and thus, how the computation proceeds is the main concern.

As shown in Fig. 2, a BPEL process exchanges messages with the partner Web service providers, and each message conforms to a particular WSDL message type defined in the `<definitions>` section. On the other hand, the `<process>` section defines how the message data are manipulated and how the control flow proceeds. The behavioral specification comes from the `<process>` section.

The correctness of BPEL programs is checked in terms of their message-type conformance and behavioral specification. The message-type conformance is a check to see whether the type of message conforms with the message type expected by the port through which the message is sent or received. This check can be considered to be a variant of type-checking[22] sometimes found in strongly-typed programming languages. This paper will not elaborate on this aspect.

Alternatively, this paper focuses on the behavioral aspect of the system because the behavioral specification plays a crucial role in distributed concurrent systems such as BPEL programs. Concurrent systems may have potential deadlocks. Such faulty behavior is not easy to uncover once a system starts executing because the error may occur non-deterministically. That is, the system exhibits such faulty behavior only in some situations and appears to execute correctly in most cases.

This paper adapts the automata-theoretic techniques for the behavioral analysis of BPEL programs. The control flow is extracted from the BPEL programs to construct a finite-state automaton which is then used to analyze how the control proceeds. This is a standard approach for theoretical and practical modeling and analysis of behavioral specifications of concurrent systems. This paper shows it to be useful for the behavioral analysis of BPEL programs.

Formal analyses are expected to show the absence of bugs, and they rely on complete but costly methods such as the interactive theorem provers. Moreover, since a sufficiently expressive language is not decidable, a sound and complete analysis would be difficult without much human intervention.

Consequently, lightweight formal methods [14] have been recognized as alternatives to the conventional heavyweight approaches. The primary concern here is not to show the absence of bugs, but to detect faulty

```
<definitions ...>
 <message name="POMessage">
   <part name="customerInfo" type="customerInfo"/>
   <part name="purchaseOrder" type="purchaseOrder"/>
 </message>
  ...
 <portType name="purchaseOrderPT">
   <operation name="sendPurchaseOrder">
     <input message="POMessage"/>
     <output message="InvMessage"/>
     <fault name="cannotCompleteOrder" ... />
   </operation>
 </portType>
  ...
 <partnerLinkType name="purchasingLT">
   <role name="purchaseService">
     <portType name="purchaseOrderPT"/>
   </role>
 </partnerLinkType>
  ...
</definitions>

<process name="purchaseOrderProcess" ...>
 <partnerLinks>
   <partnerLink name="purchasing"
                partnerLinkType="purchasingLT"
                myRole="purchaseService"/>
   ...
 </partnerLinks>
 <variables>
  <variable name="PO" messageType="POMessage"/>
  <variable name="Invoice"
            messageType="InvMessage"/>
  ...
 </variables>
 ...
 <sequence>
   <receive partnerLink="purchasing"
            portType="purchaseOrderPT"
            operation="sendPurchaseOrder"
            variable="PO">
   </receive>
   <flow>
    ...
   </flow>
   <reply partnerLink="purchasing"
          portType="purchaseOrderPT"
          operation="sendPurchaseOrder"
          variable="Invoice"/>
 </sequence>
</process>
```

Fig. 4 Purchase order (a fragment).

behavior as early as possible with the automatic analysis tool. The methods are *lightweight* in that a partial rather than a complete analysis is conducted. Although there are some opponents who argues that the lightweight approach is no more accurate than the conventional testing, it seems practical from the viewpoint of software engineering. For instance, a model checker [13] is a tool to exhaust an enormous state space to find bugs more reliably than the conventional testing techniques, yet is not a tool that can prove that a description is bug-free. This paper has studied to what degrees the behavioral analysis of BPEL can be achieved with a model checker combined with a *lightweight* abstraction technique.

Table 1 BPEL activities.

activity	function
invoke	invoking a service outside
receive	waiting for a request from outside
reply	generating the response
assign	assigning a value to a variable
sequence	sequential execution
switch	multi-way branching
while	loop
flow	concurrent execution
scope	serializability

3 Analysis of BPEL behavior

3.1 BPEL behavioral specification

3.1.1 BPEL activities

The basic computation elements in BPEL are activities to represent atomic actions. Table 1 presents a list of BPEL activities. Although the list does not cover all the BPEL activities defined in the standard document,[7] they are good for studying the essential features of a behavioral specification in BPEL. Particularly, they cover the features necessary for analyzing the four cases in the document.[7] The details will be discussed in Section 5.2.

BPEL provides three activities to exchange information with outside Web service providers: `invoke`, `receive`, and `reply`.

The `invoke` activity represents an atomic action for invoking a Web service provider via a specific `partner` link.

```
<invoke partnerLink="assessor"
        portType="riskAssessmentPT"
        operation="check"
        inputVariable="request"
        outputVariable="risk">
```

The `receive` and `reply` have similar attributes.

In addition to the above primitive activities, BPEL provides an `assign` activity for accessing variables. It also has activities to implement control flows such as `sequence` (sequential executions), `switch` (branch on conditions), `while` (repetitions), and `flow` (concurrency). The `flow` implements a flow graph that can represent concurrency. This graph displays the activities as nodes and the links as edges representing control flows.

BPEL introduces a lexical context by a `scope` activity. The lexical context defines the effective scope of variables and various handlers such as exception. However, in view of determining the control flows, a serializable scope is important. A `scope` activity can have a `serializable` attribute, which specifies that multiple concurrent accesses to the shared resources are serialized. The semantics are very similar to the standard isolation level *serializable* used in the database transaction.[7] It is defined in terms of the *strict two-phase locking* protocol.[35]

3.1.2 Concurrency in BPEL

Concurrency in BPEL is introduced by the flow activity. The idea is based on the net-oriented flow language adapted by WSFL.[19] BPEL inherits concurrency primitives from WSFL.

The specification document [7] does not fully present the operational semantics of the flow activity. This paper adapts ones from the book.[18] The semantics are based on the PM-flow [17] that has been proposed as a work-flow schema language. Moreover, they are given in a form similar to those of CPN (Colored Petri Nets).[15]

Naively, the operational semantics of BPEL are essentially rules to select activities to fire. An activity may have outgoing links connected to other activities. When the source activity completes its execution, each link is assigned a Boolean value, and is propagated downward to the connected activity. The activity is fired when all the values of the incoming links are definite and satisfy the join condition of the destination activity.

The `Purchase Order` in Fig. 3 is a simple example of using a `flow` activity in that the naive semantics can account for the concurrent execution of the three `sequence` activities.

The two of the activities in the middle part, `invoke(22)` and `receive(23)`, have an outgoing link connected to another activity, which is indicated schematically with the solid lines. Once the execution control enters into the `flow`, the three `sequence` activities can execute concurrently. However, `invoke(12)` waits for its incoming link to be enabled, the status of which is controlled by the outgoing link from `invoke(22)`. Thus, two of the concurrently executing threads are joined at the point of the activity `invoke(12)`. The `flow` terminates after all the concurrent executions terminate, and the control goes to the `reply` activity.

Fig. 5 is another example (`Loan Approval`) from the standard document.[7] The execution is more complicated than in the case of `Purchase Order`. Its BPEL source program is found in Appendix A.1 in the Appendix.

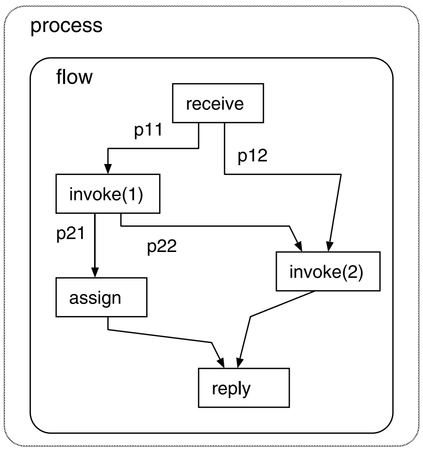


Fig. 5 Loan approval.

The `flow` has five atomic activities that are executed concurrently and have causal dependencies specified with the `links`. The `receive` activity has two outgoing control links, each of which is supposed to be set *true* depending on other variables. Fig. 5 uses variables such as `p11` and `p12` to denote the condition schematically. `Invoke(1)` on the figure's left has two links and their values determined by `p21` and `p22`. Two variables are introduced because the two links are distinct in the BPEL source program text, even though they are logically related.

Once the execution control is passed to the `flow` activity, its internal activities start executing concurrently. The `receive` activity can be fired since no condition is imposed on it. It sets the values of its outgoing links depending on the values of certain variables. Although Fig. 5 compactly illustrates that the conditions are `p11` and `p12`, they actually refer to the `transitionCondition` attribute of the following code fragment; the former concerns `p11` and the latter does `p12`.

```
<source linkName="receive-to-assess"
   transitionCondition=
      "getVariableData('request','amount')
      &lt; 10000"/>

<source linkName="receive-to-approval"
   transitionCondition=
      "getVariableData('request','amount')
      &ge; 10000"/>
```

When `p11` is *true*, `invoke(1)` is chosen to fire. Another variable `p12` is supposed to be *false* when

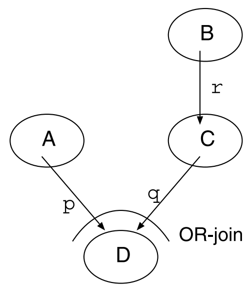


Fig. 6 Dead-Path Elimination.

p11 is *true* because the transitionCondition values are so specified.

Similarly, the values of two outgoing links of invoke(1) are determined. When p21 becomes *true*, assign is selected, after which the control finally goes to the reply activity at the bottom of the figure.

3.1.3 Dead-Path Elimination

The operational semantics of the flow are somewhat complicated by the need for DPE (Dead-Path Elimination). DPE provides a means to handle cases where an activity should be fired even when not all of its incoming links have been assigned values. Fig. 6, excerpted from the book,[18] illustrates why DPE is needed.

In Fig. 6, the result p of the activity A is *true* and the r of the activity B is *false*. Since r is *false*, activity C will never be executed. Therefore, the join condition of activity D will also never be evaluated. Activity D, however, can in principle be executed because its join condition is OR and one of its input control link p is already known to be *true*. In a word, activity D can be executed logically, but its join condition will never be evaluated in the naive semantics.

DPE provides a means to resolve such pseudo faulty situations. It starts executing when a join condition becomes *false*, or when there is an activity having a single input link with *false* only. DPE traverses the flow model downward to eliminate the pseudo-faulty situations by forcing the related join condition to be evaluated. It uses *false* values when it needs a logical calculation in the evaluation process. DPE terminates the propagation process when it reaches either an activity having a join condition or the ultimate end.

The example in Fig. 5 requires DPE if it is to be executed in the expected manner. When both p12 and p22 turn out to be *false*, the join condition of invoke(2) activity becomes *false* and the activity is never executed. Consequently, the join condition of the reply activity, actually a logical or ($\vee$) of the two incoming links, is not evaluated because one of them comes from invoke(2). On the other hand, since the outgoing link from the assign activity becomes *true*, the reply activity is logically executable. DPE can force the execution of the reply activity just as expected.

3.2 Formal analysis and abstraction

In general, abstraction is essential for the analysis of the behavioral specification. Since the analysis is done in an environment that does not have actual service providers, no concrete value or message is available. The actual values coming from the external service provider may have much effect on the control flow of the BPEL program to be analyzed.

Although the analysis should be performed without knowing the actual values, all the values that might be used should be checked. This is, however, not feasible in most cases because the number of combinations of all potential values would be huge.

As explained for the example in Fig. 5, the transitionCondition attribute in the <source> tag should be appropriately evaluated in order to have the expected executions. The example condition involves a value stored in the request variable, which is assigned in the receive activity by using a message from an external service provider. Since the actual value of the request variable is not determined at the time the analysis is conducted, it is best to say that the transitionCondition would be either *true* or *false* with equal probability. In other words, the link takes either value in a non-deterministic manner.

If such non-determinism is applied to all the transitionConditions independently, the two outgoing links of the receive activity can be either *true* or *false*, and both can be *true* at the same time in some cases. For example, in Fig. 5, both p11 and p12 may be *true* at the same time, which is not what is meant in the original BPEL application.

From the viewpoint of the formal analysis, it results in a *false negative* situation such that the detected faulty behavior is not an aspect of the original description, but one due to a poor abstraction (or lack of abstraction). Hence, this leads to the pseudo faulty situation. It is, however, not always easy to make a proper abstraction of BPEL in which no *false negative* appears. Consequently, *false negatives* are treated as a matter of degree. Regarding the example here, in order for the flow to be analyzed to a good approximation, the two outgoing links should take distinct values. Namely, when p11 is *true*, p12 should be *false*.

The approach described in the paper is to introduce auxiliary predicate variables; a predicate variable for each conditional expression. The value of the predicate variable is dependent on the BPEL variables since they constitute the concrete expressions in the BPEL program text.

As in Fig. 5, two predicate variables, *pred*1 and *pred*2, are introduced to represent the conditions on the two outgoing links from the `receive` activity as seen from the original BPEL source program.

$pred1 \triangleq$ `request.amount <  10000`
$pred2 \triangleq$ `request.amount >= 10000`

where `request.amount` is an abbreviation of

```
getVariableData('request','amount')
```

and the binary operators are used instead of the original XML expression for simplicity of the presentation;it uses < and >= instead of `&lt:` and `&ge:`. The two satisfy the following logical relationship.

$pred2 = \neg\ pred1$

Furthermore, although the value of the BPEL variable `request` is assigned in the `receive` activity, the actual value is not determined at the time of analysis. Thus, the `receive` activity should be accompanied with an auxiliary assignment that *pred*1 takes either *true* or *false* in a non-deterministic manner.

This discussion of the non-determinism of the condition is applicable to other activities involving the conditional expressions (`switch` and `while`).

4 Modeling and analysis with EFA

This paper has adapted automata-theoretic techniques to the behavioral analysis of BPEL programs. The intermediate representation has two purposes:(1) to formally define the behavioral specification of BPEL programs and (2) to clearly state the verification problem at hand.

4.1 Modeling with EFA

4.1.1 Extended Finite-state Automaton

The behavioral specification is essentially an abstract view of the system in terms of the control flow, and thus an automaton is a good tool for both the representation and analysis. BPEL, however, has language constructs to express data flow aspects:some activities exchange messages with the partner service providers via partner links, and the incoming messages are stored in the variables. In addition, variables and links may affect the control flow: variables may appear in expressions of the condition in `switch` and `while`, and may also be used in the condition to fire particular links in the `source` element. Taking into account some notion of variables is essential and EFA would be a basis for the representation.

Formally, an EFA (Extended Finite-state Automaton) M is a 7-tuple: $M = \langle \mathcal{Q}, \Sigma, \mathcal{V}, \rho, \delta, q_0, \mathcal{F} \rangle$.

$\mathcal{Q}$: Finite Set of Location Points
Σ: Alphabet including Symbols below
  P ! X : Output Action Designator
  P ? X : Input Action Designator
  ϵ : Internal Action Designator
$\mathcal{V}$: Finite Set of Variables
ρ : Variable Map $\mathcal{Q} \rightarrow 2^{\mathcal{V}}$
δ : Transition Relation $\mathcal{Q} \times \mathcal{A} \times \mathcal{Q}$
  $\mathcal{A}$: Transition Action $\Sigma \times \theta \times \mathcal{G}$
    θ : Variable Update Functions
    $\mathcal{G}$: Guard Condition
$q_0 \in \mathcal{Q}$: Initial Location
$\mathcal{F} \subseteq \mathcal{Q}$: Finite Set of Final Locations

An EFA M is basically a finite state automaton, but has variables $\mathcal{V}$ which are assigned by the update functions (θ) and used in the expressions for the input/output action designators (Σ) and the guard conditions ($\mathcal{G}$). The set of variables at a particular location point is obtained with the variable map (ρ).

The transition relation (δ) is a triple relating a source and a target location point ($\mathcal{Q}$) with a transition action ($\mathcal{A}$), which is also a triple consisting of an alphabet (Σ), a variable updating function (θ) and a guard ($\mathcal{G}$). The operational meaning of a transition relation is that the current location of the source is changed to the target when the specified action is taken if the accompanied guard condition is *true*. The variable updating function is executed during the course of the transition, which assigns a new value to the specified variable in the target location point. The variable updating function is actually a set of simultaneous assignments.

Each alphabet in Σ may take one of the three forms. The two forms, P ! X and P ? X, are in relation to communications with the external environment or automaton, while ϵ is an internal action. Specifically P ? X and P ! X are respectively input and output action designators. Operationally, the input action is receiving a message coming from P and setting X to the value in the message. The output action is sending a message with the value of X to the communication channel P.

Fig. 7 diagrammatically represents a fragment of EFA focusing on a particular transition. This example shows that the transition of the interest is the one from

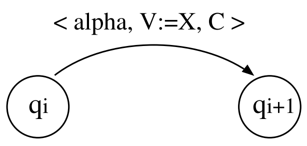


Fig. 7 Diagram representation.

the location point q_i to q_{i+1}. The label attached to the transition arc refers to a triple consisting of an alphabet symbol (`alpha`), a variable assignment(`V:=X`), and a guard condition (`C`). Underbar (`_`) is used when some of the components in the triple are irrelevant.

An asynchronous product of two EFAs is defined in a standard way. Given two EFAs M_1 and M_2, the asynchronous product is another EFA

$M = \langle \mathcal{Q}, \Sigma, \mathcal{V}, \rho, \delta, q_0, \mathcal{F} \rangle$ where
$\mathcal{Q}$: the Cartesian product $\mathcal{Q}_1 \times \mathcal{Q}_2$
Σ : the union set $\Sigma_1 \cup \Sigma_2$
$\mathcal{V}$: the union set $\mathcal{V}_1 \cup \mathcal{V}_2$
ρ : the variable map $\mathcal{Q}_1 \times \mathcal{Q}_2 \rightarrow 2^{\mathcal{V}_1 \cup \mathcal{V}_2}$
δ : the set of tuples $((q_1, q_2), \alpha, (p_1, p_2))$ such that
$\exists (q_i, \alpha, p_i) \in \delta_i$ and $q_j = p_j$ and $i \neq j$
q_0 : the tuple (q_{10}, q_{20})
$\mathcal{F}$: the subset of those elements of $\mathcal{Q}$ that satisfy the condition $\forall (q_1, q_2) \in \mathcal{F}$ and $\exists q_i \in \mathcal{Q}_i$

As usual, two concurrent EFAs are merged by taking their asynchronous product. It is the same as adapting the interleaving semantics of concurrency.

4.1.2 EFA for BPEL

A customized version of EFA, M_{BPEL}, is described below to take into account of the features specific to BPEL language constructs.

First, the communication channels appearing in the input / output designators are used to represent `partnerLinks` connected to the external service providers.

Second, the variables $\mathcal{V}$ are partitioned into three non-overlapping sets.

$\mathcal{V}$: Finite Set of Variables. $\mathcal{V}_B + \mathcal{V}_L + \mathcal{V}_P$
$\mathcal{V}_B$: Finite Set of BPEL Variables
$\mathcal{V}_L$: Finite Set of Link Variables
$\mathcal{V}_P$: Finite Set of Predicate Variables
$\rho_B : \mathcal{Q} \rightarrow 2^{\mathcal{V}_B}$
$\mathcal{G}_{PL}$: Guard Condition

$\mathcal{V}_B$ denotes a set of variables appearing explicitly in the source BPEL program. These variables are extracted from the `<variables>` tag in the `<process>` description. An additional variable map ρ_B may be used to obtain a set of BPEL variables at a particular location point.

$\mathcal{V}_L$ is a set of link variables, each of which corresponds to a `<link>` introduced in the `<links>` tag in the `flow` activity. The `<link>` specifies the control flow among the concurrently executing activities in the `flow` activity. Each link can be regarded as a Boolean variable. It is set *true* when the control flow exists.

$\mathcal{V}_P$ is a set of Boolean-valued predicate variables, but does not appear explicitly in the BPEL program. Each variable corresponds to a predicate that represents the conditional expression appearing in a `switch`, `while`, or `transitionCondition` of `<source>`. For the example in Section 3.2, *pred*1 and *pred*2 are the predicate variables that are the abstractions for the conditional expressions.

So far, the domain of the data, of which the element value is assigned to a variable, has not been made explicit. The variables in $\mathcal{V}_P$ are Boolean-valued, and so are those in $\mathcal{V}_L$ if the naive semantics are adapted. The domain of the link variables ($\mathcal{V}_L$) should be extended for proper handling of DPE, which will be discussed in Section 5.1.3.

The BPEL variables in $\mathcal{V}_B$ are application-specific and may have values in an infinite domain. Note that it is not feasible for the analysis to account for all the values when the domain is huge or infinite.

Thus, it is assumed in this paper that a BPEL variable in $\mathcal{V}_B$ takes on a value from a finite set consisting of only two elements, *definite* and *undefined*. This is equivalent to introducing symbolic abstractions.[6] Given a constant of interest *c*, the abstraction function is chosen so that

$$abs_c(x) = \begin{cases} \textit{definite} & \text{if } x = c, \\ \textit{undefined} & \textit{otherwise.} \end{cases}$$

The guard condition is also customized to be $\mathcal{G}_{PL}$, which is actually a predicate involving variables only from either $\mathcal{V}_L$ or $\mathcal{V}_P$, none from $\mathcal{V}_B$. Since they are essentially Boolean-valued, the guard condition expression *g* is either a simple propositional expression or the equality (or the inequality) of two operands. Therefore the EFA for BPEL in this paper is data-independent which is known to have nice properties for verifications.[39]

4.2 Verifications

The analysis is based on a state explosion approach for verifying the behavioral specification of BPEL programs expressed in terms of the EFAs. A number of definitions are needed to explain the verification problem at hand.

The informal notion of how a BPEL program execution proceeds can be captured by considering a *run*. A *run* is an infinite sequence of location points that an EFA generates and is represented as

$$\sigma^\omega = q_0 q_1 q_2 \ldots$$

where $\forall q_i \in \mathcal{Q}$. The stutter extension rule [13] is assumed to represent a finite run as an infinite one. And an accepting run is defined in a standard way as

$$\exists q_f \bullet q_f \in \mathcal{F} \wedge q_f \in \sigma^\omega.$$

It is a successful execution path of the BPEL program.

The first verification problem concerns *reachability* and is often referred to as a check for deadlocks. It checks whether or not a BPEL program will stop in an accidental manner. Specifically, the check is whether there is a run that is not an accepting one.

The second verification problem is related to properties expressed in terms of LTL (Linear Temporal Logic), and the standard model checking algorithm can be used in the analysis. The LTL formula f can have the temporal operators `[]` (always), `<>` (eventually), and `U` (strong until), in addition to the standard logical connectives ($\wedge, \vee, \neg, \rightarrow$). The semantics of the temporal operator are given in terms of the run in a standard manner.

$$\begin{array}{ll} \models []f_1 & \Leftrightarrow \sigma^{\omega} \models f_1 \\ \models <>f_1 & \Leftrightarrow \exists k \le 0, \sigma^{k} \models f_1 \\ \models f_1 \,\mathtt{U}\, f_2 & \Leftrightarrow \exists k \ge 0 \bullet \sigma^{k} \models f_2 \text{ and} \\ & \qquad 0 \le \forall j \le k \bullet \sigma^{j} \models f_1 \end{array}$$

Here, a finite prefix of a run of the length k is represented as

$$\sigma^{k} = q_0 q_1 q_2 . . . q_{k\text{-}1}.$$

The LTL formula takes an atomic proposition P, which is a boolean expression formed by using information at each location point. The expression may refer to BPEL variables since the EFA has a variable map ρ_B in its definition making it possible to obtain BPEL variables at each location point. The properties that can be expressed, however, concern only reachability of the data tokens because the data domain is abstracted.

Another property of interest is to specify which location point the execution goes through. This can be expressed by using an atomic proposition asking whether a run contains a specific q_i defined for each location point.

A simple LTL formula for a process starting with `receive` and eventually reaching `reply` can be written as below

```
[]((receive) -> <>(reply)),
```

which the examples in Fig. 3 and 5 satisfy.

4.3 BPEL to EFA

This section describes a scheme for translating from a BPEL activity to an EFA fragment. The diagrammatic representation shown in Fig. 7 is used since it is more intuitive than the presentation in text.

4.3.1 Atomic activities

The translations of atomic activities are mostly straight forward. Since an EFA already has notions of send-receive channel communications and variables, activities involving communications with external service providers (`reply`, `receive`, and `invoke`) and variable assignments (`assign`) are easily translated into an appropriate EFA fragment.

- receive

```
<receive partnerLink=L portType=T
          operation=Op variable=V>
```

$$\begin{tikzcd} q_i \arrow[r, bend left, "< L\ ?\ Op(V)\ ,\ \_\ ,\ \_ >"] & q_{i+1} \end{tikzcd}$$

The `receive` activity is essentially for receiving an appropriate message from a communication channel. The channel is specified by the `partnerLink`, and the message is constructed from the operation and variable as `Op(V)`. Operationally, the transition is enabled when a message of the form `Op(A)` is transmitted through channel `L`, and the value `A` embedded in the message is assigned to the variable `V`.

- reply

```
<reply partnerLink=L portType=T
        operation=Op variable=V>
```

$$\begin{tikzcd} q_i \arrow[r, bend left, "< L\ !\ Op(V)\ ,\ \_\ ,\ \_ >"] & q_{i+1} \end{tikzcd}$$

The `reply` activity is interpreted similarly. The transition is enabled when a message `Op(V)` is sent through `L`.

- invoke

```
<invoke partnerLink=L portType=T
      operation=Op inputVariable=IV
      outputVariable=OV>
```

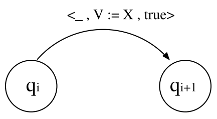
$$\begin{tikzcd} q_i \arrow[r, bend left, "< L\ !\ Op(IV)\ ,\ \_\ ,\ \_ >"] & q_{i+1} \arrow[r, bend left, "< L\ ?\ Return(OV),\ \_\ ,\ \_ >"] & q_{i+2} \end{tikzcd}$$

The `invoke` activity takes two forms; the first one shown above is similar to a remote-procedure call. Since it is meant for a synchronous communications and is roughly said to be a combination of a message send and a receive. The EFA fragment

consists of two consecutive transitions. The transition corresponding to the receive expects a message of the form `Return(X)`.
The constant symbol `Return` denotes that the incoming message contains a return value of the invocation.
The second `invoke` activity does not use `outputVariable` in a one-way communication, and is basically the same as `reply`.

- assign

```
<assign>
 <copy>
  <from variable=X/>
  <to variable=V/>
 </copy>
</assign>
```



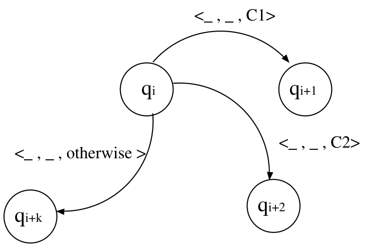
The `assign` activity represents single or multiple variable assignments. It has several variations as a form of the `<from>` tag such as a variable dereference, an expression, or a literal value.
The `<to>` tag may refer to a `partnerLink`, which is considered as a message send, and thus the translation is similar to the case of `reply`.

4.3.2 Control aspects

Complex activities forming control sequences may make use of the guard condition of an EFA.

- switch

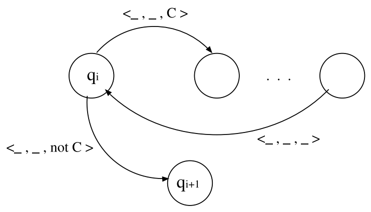
```
<switch>
 <case condition=C1> ... </case>
 <case condition=C2> ... </case>
 <otherwise> ... </otherwise>
</switch>
```



The `switch` activity is a multi-way conditional branching control and has a `otherwise` branch. Actually, `otherwise` should be an appropriate expression such as $\neg C_1 \wedge \neg C_2$.

- while

```
<while condition=C>
...
</while>
```



The `while` activity introduces an iteration and requires a loop structure to represent the repetition.

4.3.3 Flow

The flow activity's concurrent computation aspect should be taken into account in the translation. Basically, all the direct sub-activities enclosed in a `<flow>` tag are considered as concurrently executing entities. Each sub-activity becomes an EFA, and they all are composed by taking their asynchronous product, which means that the method used here adapts the interleaving semantics of the concurrency. It, however, requires synchronization machinery to account for the semantics in that the flow activity terminates when all its enclosing sub-activities terminate.

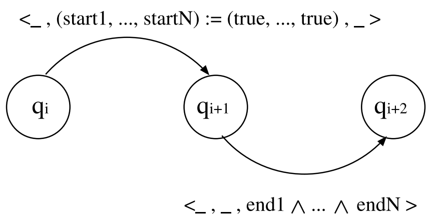
First, the top `<flow>` becomes a part of a *main* EFA and each sub-activity appearing directly in the `<flow>` is translated into a *sub* EFA. The *main* EFA transfers execution control to all the concurrently executing *sub* EFAs and waits for their completions. Second, the EFAs are composed into one EFA by taking the asynchronous product. Since the asynchronous product of automata is a standard operation, a sketch of translating the `<flow>` and its sub-activities is shown below.

For example, in `Purchase Order` (Fig. 3), the flow activity has three direct sub-activities, each of which becomes a *sub* EFA automaton. The flow is also a component of its enclosing `<sequence>` together with a `<receive>` and a `<reply>` as a kind of *brother* subsidiaries (Fig. 4). These three subsidiaries of `<sequence>` becomes the *main* EFA.

- BPEL source fragment

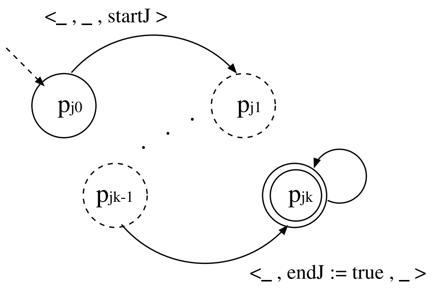
```
<flow>
 < -- sub-activity-1 -- >
 ...
 < -- sub-activity-N -- >
</flow>
```

- a *main* EFA fragment



<_ , (start1, ..., startN) := (true, ..., true) , _ >
qi
qi+1
qi+2
<_ , _ , end1 ∧ ... ∧ endN >

- a *sub* EFA fragment

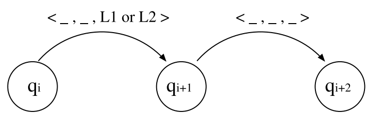


- Auxiliary control variables
  In order for both the *main* and *sub* EFA components to synchronize their executions, auxiliary control variables must be introduced.
  For each *sub* EFA, two boolean variables `startX` and `endX` are defined. The variable `startX` is set *true* when the execution control is entered into the `flow`, which is described in the *main* EFA automaton.
  Each *sub* EFA automaton has an adequate guard condition to refer to `startX` and waits for it to become *true*. The *sub* EFA starts executing when `startX` is set *true*.
  Upon its completion, each *sub* EFA sets `endX` *true*. The *main* ensures that all the sub-activities are certainly terminated by using the appropriate guard condition on `endX`s, i.e. logical-and of all `endX`s.

Furthermore, an atomic activity enclosed in a `flow` may be synchronized in regard to its incoming links. An activity may also have `<source>` tags with the `transitionCondition`. For simplicity, the example below shows the case without DPE.

- synchronization of the incoming links

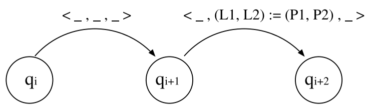
```
< -- activity -- >
  <target linkName=L1>
  <target linkName=L2>
</ -- activity -- >
```



The enclosing activity `< -- activity -- >` may have a particular synchronization condition as its `joinCondition` attribute. The condition is interpreted as a *logical-or* when the attribute is not explicitly specified. The EFA fragment above shows this default case of the logical-or ($\mathtt{L1} \vee \mathtt{L2}$).

- `<source>` tags with `transitionCondition` (abbreviated to `tC`)

```
< -- activity -- >
  <source linkName=L1 tC=P1>
  <source linkName=L2 tC=P2>
</ -- activity -- >
```

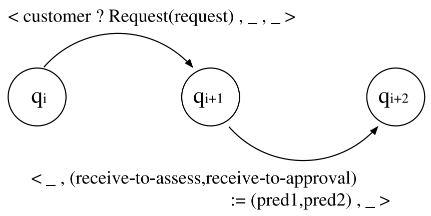


The `<source>` tag sets its attribute `linkName` to *true* when `tC` is *true*, and *false* otherwise. This means that the `linkName` is always set to the value of `tC`, and thus everything can be taken care of with the variable update function.

Although the EFA with an appropriately chosen domain of $\mathcal{V}_P$ can deal with DPE, its EFA fragment would be lengthy and not compact enough to show here. The discussion on how DPE is handled is postponed to Section 5.1.3.

4.3.4 Example translation

Below is an example translation of a BPEL activity. The `receive` activity in Fig. 5 may be represented as



< customer ? Request(request) , _ , _ >
qi
qi+1
qi+2
< _ , (receive-to-assess,receive-to-approval) := (pred1,pred2) , _ >

where *pred*1 and *pred*2 are predicate variables as introduced to denote the transition conditions in an appropriate manner (see Section 3.2).

5 Analysis with SPIN

5.1 EFA to Promela

This section describes the method of using the SPIN model checker for the representation of the EFA model and the analysis. The basic idea is simply to translate an EFA into a Promela source program, the specification language for SPIN. Promela adapts the computational model of channel communicating finite-state automata with variables.

5.1.1 Translation of basic features

The translation is mostly straight-forward since Promela is expressive enough to represent control structures, channel communications, and variables as well as automata. The translation is achieved as follows.

- An EFA automaton M becomes a Promela process.

- The communication channel P appearing in the input or output action designator is translated into a Promela channel. Since P denotes a `partnerLink` in BPEL, the name of the Promela channel *name* is taken from the `name` attribute. The Promela channel declaration takes into account the *type* of message exchanged.

  `chan` *name* `= [0] of {mtype, short};`

  where `mtype` is an enumeration type describing the `operation` and the second argument carries the data value, actually a data token.

- The variable $\mathcal{V}$ is translated into a (global) variable in Promela. Specifically, predicate variables $\mathcal{V}_P$ are boolean Promela variables initialized as *false*.

  `bool` *name* `= false;`

  Link variables $\mathcal{V}_L$ are also basically boolean, but a slightly different encoding is used in order to deal with DPE. Section 5.1.3 describes in detail how DPE is handled.

- Encoding the transition relation δ is the most interesting part of the translation.

  - The control aspect is directly encoded with the Promela control language constructs:an unconditional transition is represented as a Promela sequencing operator (`;`).

  ```
  action1 ;
  action2 ;
  ...
  ```

  Conditional branching in EFA, making use of the guard condition $\mathcal{G}$, turns into the following.

  ```
  if
    ::cond1->...
    ::cond2->...
    ...
    ::else->...
  fi;
  ```

  Repetitions are easy to represent in Promela since the language provides the loop construct.

  ```
  do
   :: condition ->...
   :: else -> break
  od;
  ```

  - Input or output action designators are Promela channel operations. Both P ? X and P ! X are translated into Promela counterparts. The channel P is defined as a Promela channel as discussed above.

    channelName `?` *Op*(*variableName*)`;`

    and

    channelName `!` *Op*(*variableName*)`;`

  - The variable update function θ denotes a set of simultaneous assignments of multiple variables. Atomicity is introduced in its Promela translation.

    ```
    atomic{
       variableName = newValue ;
    ...
    }
    ```

Two more remarks would be appropriate here. First, in regard to the EFA representation of the flow activities, Section 4.3.3 indicated that the concurrency aspects are understood in terms of interleaving semantics and that the component EFA automata are merged by the asynchronous product operation. However, the

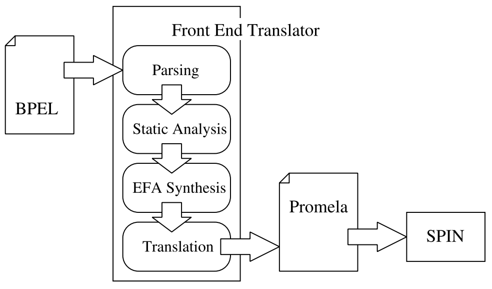


Fig. 8 Tool architecture.

product operation is not needed to perform at the EFA level. Each component automaton is independently translated into a Promela process description, and the SPIN tool is responsible for the *merging*.

Second, in order to analyze the behavior, the Promela model should be *closed*. A closed model has Promela processes to simulate the environment in which the target BPEL process is supposed to execute. Actually the environment contains all the service providers with which the BPEL process has communication via the appropriate partner links.

However, defining the environment is not as difficult as it seems to be. When the partner BPEL process source program is available, the environment model for the analysis can be obtained by means of the method described in this paper. When it is not available, the environment Promela process is such that it is only concerned with the externally visible sequence of exchanged messages.

5.1.2 Abstraction and static analysis

As discussed in Section 3.2, abstraction is necessary to obtain an EFA from a BPEL program. Static analysis is needed in order to introduce appropriate predicate variables ($\mathcal{V}_P$). As briefly presented by using the example in Section 3.2, the analysis is basically a define-use chain (du-chain) of variables having effects on the control flow. Furthermore, a single predicate variable is introduced for each conditional expression. The proposed analysis tool has four steps as shown in Fig. 8: Parsing, Static Analysis, EFA Synthesis and Translation into Promela.

Although the example in Fig. 5 is simple, the translation of the `receive` activity might be illustrative. It requires two predicate variables to be logically related. The fragment relevant to the discussion here is shown below. It uses abbreviations such as < and >= instead of `&lt:` and `&ge:`.

```
1: <receive partnerLink="customer"
     operation="request"
     variable="request">
2: <source linkName="receive-to-assess"
     tC="request.amount < 10000"/>
3: <source linkName="receive-to-approval"
     tC="request.amount >=10000"/>
     </receive>
```

The Promela code fragment looks like,

```
  Customer? Request(request);
  if::pred1 = true::pred1 = false fi;
  pred2 = ! pred1;
  receiveToAssess = pred1;
  receiveToApproval = pred2;
```

The variable `pred1` denotes the expression

```
  request.amount < 10000,
```

and the variable `pred2` is chosen so as to correspond to

```
  request.amount >= 10000.
```

Both predicate variables depend on the variable `request`.

The location where `pred1` takes a new value (a def-location)is at the `receive` activity (line number `1:`) where it is assigned a new value. In addition, the location where it is accessed (a use-location) is at the first `<source>` tag with `"receive-to-assess"` (at `2:`), in which `request.amount` is accessed to determine the value of the link. The def-location of `pred2` is the same as that of `pred1` (at `1:`), while its use-location is at the second `<source>` tag with `"receive-to-approval"` (at `2:`).

The next thing is to determine the value of `pred1`; note it is best to say that the variable `pred1` would be either `true` or `false` because the `request` value is undefined at the time of analysis. A code fragment to set the variable `pred1` either one non-deterministically is inserted at the def-location. Once the value of `pred1` is determined, the search algorithm used in the model checking method can explore all the possible combinations.

A standard technique of static program analysis for the def-use chain can determine both the def-location and use-location. However, it requires additional machinery to conclude that the two predicate variables are related as

```
  pred2 = ! pred1.
```

Currently, a simple syntactic pattern matching is employed to derive such logical relationships between the predicate variables. Such a syntactic matching method sacrifices the completeness but requires less computation to find appropriate relationships. If the BPEL program has expressions requiring more complicated logical inference, the current method does not derive correct logical relationships and leaves all the predicate variables as if they are independent. This implies that such an analysis may have false negatives, which is the limitation of the lightweight analysis technique adapted in this paper.

Most of the translation of the last step is presented in Section 5.1.1, however, special care should be taken

```
<variables>
 <variable name="shared" .../>
</variables>
<flow>
  <links> <link name="foo"/> </links>
  <scope variableAccessSerializable="yes">
    <sequence>
     <assign> ... </assign>
     <empty> <source linkName="foo"/> </empty>
     <assign> ... </assign>
   </sequence>
  </scope>
  <scope variableAccessSerializable="yes">
    <sequence>
     <assign> ... </assign>
     <epmty> <target linkName="foo"/> </empty>
     <assign> ... </assign>
   </sequence>
  </scope>
</flow>
```

Fig. 9 Deadlocked serializable scopes.

in regard to DPE (Section 3.1.3) and the `scope` with the `serializable` flag *true*.

5.1.3 Handling DPE in Promela

In order to handle DPE properly, the logical values should be extended to have *forced* in the domain of $\mathcal{V}_L$. The value is used as an indicator that DPE has activated.

The *forced* value is generated and flows downward exactly when the DPE starts. If the activity has a join condition, the *forced* value is interpreted as *false* in the evaluation of the condition. If the activity does not have a join condition, the *forced* value ensures that the activity is not executed but is instead flowed down along the output control link of the activity.

The Promela program first changes the *forced* value to be *false* and then the join condition is checked. The activity body is executed if the condition is satisfied, and is skipped otherwise. Furthermore, new link values are generated. The new values are determined by the `tC` in the `<source>` tag if the activity is executed normally, and are set as *forced* in the case of DPE. A Promela fragment might look like below.

```
∀ link ∈ IncomingLinks •
     link = ((link == forced) -> false : link);
if
   :: joinCondition (IncomingLinks) ->skip
   :: else -> goto SkipBody
fi;

activity body ;
∀ link ∈ OutgoingLinks • link = tC ;
goto EndOfActivity ;
SkipBody:
∀ link ∈ OutgoingLinks • link = forced ;
EndOfActivity: skip ;
```

5.1.4 Serializable scope

A serializable scope specifies that multiple concurrent accesses to the variables inside the scope are serialized. An access sequence, i.e. a schedule, is generated with the strict two-phase locking protocol.[7] The following describes a simple scheme to use a mutex lock for the behavioral analysis of the BPEL program with a serializable scope.

The Promela program introduces a mutex for each such serializable scope for mutually exclusive accesses to the variables in the critical region. According to the standard coding style, the Promela code fragment to make accesses to shared variables looks like

```
atomic{mutex == free -> mutex = busy};
      critical region to access variables
atomic{mutex = free}
```

Although it seems to work well with an adequate interpretation of the serializable scope, a simple example resulting in a deadlock situation can be constructed. Actually, BPEL provides more than one means to express serialization of accesses to shared resources or shared variables. Thus a careless combination may lead to faulty behavior.

Fig. 9 is one such example of a BPEL program fragment, which uses both the serializable scope and the link to control accesses. In the example, both `assign` activities have write accesses to the variable `shared`. A deadlock appears when the second `scope` precedes the first one to obtain the mutex. The `empty` having an incoming link waits for it to be enabled. The link, however, is never enabled by the `source` in the first `scope`, which cannot be executed because the mutex is held by the second one.

Although it is not difficult to remove the deadlock, the example indicates the importance of behavioral analysis for a BPEL program using a serializable scope.

5.2 Four analysis cases

Table 2 summarizes the experiment to test for deadlocks. The examples used in the experiment were taken from the standardization document.[7] In each case, an appropriate *environment* Promela description was introduced as *closed*. The environment process was manually constructed so that it would not have much effects on the size of the state space.

Table 2 Four cases.

	Name	BPEL Features	States	Analysis Methods
1	Purchase Order	variable, flow	249	basic technique
2	Shipping Service	switch, while	21	P-Variables
3	Loan Approval	flow	3,516	P-Variables, DPE
4	Auction Service	multiple start	57	basic technique

```
receive shipOrder
switch
  case shipComplete
    send shipNotice
  othewise
    itemsShipped := 0
    while itemsShipped < itemsTotal
    itemsCount := internally generated
    send shipNotice
    itemsShipped := itemsShipped + itemsCount
```

Fig. 10 Shipping service outline.

```
receive shipOrder
p1 := either true or false
switch
  case p1
    send shipNotice
  othewise
    itemsShipped := 0
    p2 := either true or false
    while p2
      itemsCount := internally generated
      send shipNotice
      itemsShipped := itemsShipped + itemsCount
      p2 := either true or false
```

Fig. 11 Shipping service abstraction.

Although its exact value is not significant, the size of the state space (#States) roughly shows how complicated the analysis will be. As discussed in Section 5, because the translation makes full use of the Promela language constructs, the state space is small in most example. The third example, Loan Approval, contains five activities to be executed concurrently, and its state space is the largest of the examples due to the interleaving semantics of the concurrency. The experiment used the latest version of the SPIN tool executed under Windows/XP operating on a recent laptop computer. All the analysis terminated almost instantaneously, which shows that the translated Promela program did not have any trouble from the viewpoint of runtime cost.

Purchase Order Purchase Order is the initial example in Section 6.1 of the document [7] to illustrate the most basic structures and some of the fundamental concepts of the BPEL language. Fig. 3 schematically shows the overall structure of the control flow, and Fig. 4 shows a fragment of the BPEL program. Although the example has three concurrent `sequences`, the control is simple: each of the three activities executes sequentially and there are only two simple unconditional control dependencies. Consequently, the state space is not large.

Shipping Service Shipping Service is an abstract process to describe a rudimentary shipping service. Fig. 10 shows an outline of the Shipping Service process. It uses `switch` and `while` activities to implement adequate controls with variables such as "items-Shipped" and "itemsTotal". Since their values are not determined, abstraction is needed for the analysis. The technique with the predicate variable (or P-variable) is used so that the branch conditions are chosen non-deterministically. All the possible conditional branch paths are explored.

Fig. 11 is an abstraction of Fig. 10. It introduced two predicate variables *p1* and *p2*; *p1* represented "ship-Complete" and *p2* stood for "items-Shipped < items Total". Note that a code fragment describing the assignment of the predicate variable was inserted where the original variables related to the predicate variable were updated. In this particular example, the abstraction was successful, which meant that no false negatives were reported.

Loan Approval Loan Approval makes use of BPEL concurrency with the `flow` activity. As shown in Fig. 5 and the Appendix, all the activities are contained within a `flow` activity, and their potentially concurrent behavior is staged according to the dependencies expressed by the corresponding `links`. The transition conditions attached to the `source` elements of the links use variables to decide which outgoing links should be activated. DPE is essential to propagate the information indicating that some of the activities are not fired and thus skipped. This example also requires that the variables affecting the evaluation of the transition conditions to be abstracted.

Some other interesting characteristics of the example may be expressed as an LTL formula. The property that either `assign` or `invoke`, not both, is executed can be checked.

```
[](receive ->
       (<>assign && []!invoke(2)
       || <>invoke(2) && []!assign))
```

which reads such that the process starting with receive eventually reaches assign (or invoke(2)) and there is no invoke(2) (or assign) activity at all. The number of states to explore was 16,518;the SPIN model checker analyzed the model almost instantaneously.

Auction Service Auction Service is an example with multiple start activities in which more than one receive activities exist in a flow. It has no new feature to be analyzed. A manual translation, however, can reduce the size of the state space to about 25%. This is because a Promela fragment to represent the multiple starts can be better optimized than the general translation scheme discussed in this paper. Unfortunately, mechanical optimization does not seem easy since it requires annotations to the BPEL program to indicate the existence of multiple starts.

6 Discussions

6.1 EFA as a representation

This research adapted the automata-theoretic techniques for the behavioral analysis of BPEL programs, and introduced an EFA as the intermediate representation. The EFA has two purposes: (1) to define the behavioral specification of BPEL programs and (2) to clarify the verification problem at hand.

Direct translation from a BPEL program to generate Promela source text might be possible in principle. Since Promela is very expressive and can even describe an automaton with an infinite state space, it may blur the definition of the behavioral specification of the BPEL application program. A precise intermediate formalization is necessary.

Furthermore, with the intermediate representation, the translation of the BPEL program into Promela program can be staged. It is often the case that the correctness of the translation is hard to confirm. By using the staged approach, the gap between the source and target representation can be made as small as possible, which aids inspection of the translation process.

The behavioral specification is essentially an abstract view of the system in terms of the control flow, and thus a finite-state automaton is a good tool for both representation and analysis. BPEL, however, has variables, and their values have an effect on the behavioral specification. The automaton should be augmented with a notion of variables. Consequently, EFA was adapted as described in Section 4.1. Moreover, the EFA for BPEL is basically data-independent,[39] and the definition is mostly borrowed from the formulation in.[30] Although a number of variables are involved, data-independent automaton can be finite, which is amenable to state explosion techniques. It is thus important to know that a behavioral specification generated by a certain non-trivial subset of BPEL can be represented with a data-independent automaton.

However, the work reported here is not unique in its adaptation of EFA. Using such a model would be a standard approach for the behavioral analysis. Fu et al. [11] use guarded automata for representing the behavioral specification of BPEL. Their guarded automaton model seems essentially the same as the EFA in the present paper. Both can use variables, and the transition may have guard conditions. However, Fu et al. do not discuss the relationship between the guarded automata and other formalisms such as the data-independent automaton. Wombacher et al. [38] use an Annotated DFA to represent the message interaction sequences extracted from a BPEL program. Their motivation is to use the automaton for service recovery queries, and the model is less expressive than EFA or guarded automata.

6.2 Lightweight method for abstraction

The abstraction in this paper refers to the technique using a static analysis of a BPEL program to extract the control structure of with an adequate level of precision.

The analysis of the Shipping Service and Loan Approval examples need abstractions to obtain proper results. In general, the abstraction, or approximation, is important for the formal analysis with model checking techniques.[6] This paper adapted an abstraction method using the predicate variable (P-variable) as discussed in Section 3.2. All conditional branchings depend on the equality check (inequality check) of the P-variables, which is consistent with the choice of the data-independent EFA as the intermediate representation.

Since BPEL is basically a language for distributed collaboration of concurrent processes, it is not surprising that several researchers on BPEL analysis employ a process algebra formalism such as CCS or CSP. However, the works to use the process algebra study neither abstraction nor DPE. For example, a conditional branch is simply translated as a non-deterministic choice. This is basically the same as the case that both p11 and p12 take *true* at the same time, which is discussed in Section 3.2's example of an inadequate situation. As a result, an analysis embodying a process algebra formalism may have more *false negatives* than the analysis in this paper.

An abstraction technique often used with model checking is *predicate abstraction*.[12] When more than one variable is involved in an expression to determine control flows, all the variable values should be considered, which may result in a combinatorial explosion. The predicate abstraction technique introduces a predicate or a boolean variable for each conditional expression. The *P-variable* of this paper falls into this category.

Although predicate abstraction provides a systematic way of obtaining a good approximation of the control flow, choosing the appropriate predicates is not easy.[34] This is because the choice requires the knowledge of both the description to be verified and the property to check.

In order to obtain a set of appropriate predicates automatically, SLAM [1] adapts an iterative method to refine and obtain predicates that can discriminate execution paths in enough detail. SLAM aims to analyze programs written in C, and incorporates a symbolic evaluator for C programs, which is used in the refinement process. SLAM further re-constructs the state machine whenever it introduces new predicates. It employs a theorem prover for checking the validity of the constructed state machine. This sometimes results in a drastic change in the structure of the state-machine so that the description is hard to trace back to the original program.

Modex [13] takes a modest approach that has less automation, but allows a translated program to be structurepreserving. Since the traceability between the original program and the description for the analysis is good, it is not difficult to extract from the analysis results the information to *debug*.

The P-variable approach of this paper is also meant to be structure-preserving. As shown in Fig. 8, the translation starts from BPEL program, which is modeled as EFA after an appropriate static analysis, and results in a Promela description. The P-variable is introduced at a control location point, which causes the structure of the generated Promela to reflect the structure of the original BPEL program faithfully.

The current method to handle the P-variables is not complete in that it sometimes fails to derive their correct logical relationships. Instead of introducing the adequate relationships between them, it regards them as if they are independent. Such P-variables will result in false negatives.

However, such an inadequacy is, in some sense, a compromise between the degree of precision and computational cost, which should always be considered in the case of the lightweight formal methods. Since the four examples can be precisely analyzed, the approach presented here seem satisfactory, although higher approximation would be always desirable.

6.3 Coverage of BPEL

Although it covers most of the interesting features of behavioral specifications, the approach of this paper is not complete for the formal analysis of BPEL application programs.

First, this paper has focused on the language constructs in Table 1 only: while `<empty>` and `<terminate>` are easy to introduce. Among the other constructs, `<pick>` would not be difficult to introduce. This paper, however, did not consider <pick> because its semantics are problematic when combined with DPE.[3]

Second, this paper focused on the behavioral specification and did not consider the data aspect of BPEL programs. It is important to have some notion of type conformance checking such as port type and message type, and there is research on this topic.[11][22]

Third, BPEL is a large language that has more features than those covered in this paper. These include correlation sets, various error handlers (compensation handlers, fault handlers, event handlers). This paper ignored these features though they are used in the BPEL application programs.

The handlers may affect much of the behavioral specifications of BPEL applications that deal with exceptional cases, and the analysis method should include such features. Unfortunately, as discussed in the literature,[21] the semantics of BPEL handlers are not yet completed and a clear and precise definition is lacking. Its inclusion in the analysis method will be an important future work.

6.4 Related work

The earliest work on the formal verification of Web service flows can be traced back to the paper by S. Narayanan and S.A. Mcllraith who employ Petri-net for the automatic analysis.[28] They, however, do not address languages relating to the Web service standardization activities.

The first proposals that aim at standardization are WSFL and XLANG in May 2001. And S. Nakajima mentions the basic idea of using model checking techniques for the analysis of WSFL.[23][24] Since WSFL provides a kind of general-purpose activity and is based solely on flows and links to represent control aspects, it is not easy to have Promela descriptions that lead to a state space small enough for an analysis to be practical. A simple example in [19] reached almost a million states.[25]

Although WSFL is obsolete, the basic technique to deal with concurrency including DPE is still applicable to BPEL.[27] The research of this paper employs the same technique to handle DPE for the case of BPEL, and deals with the other techniques that would be needed to analyze BPEL application programs by using a model checker. Thanks to the various advanced BPEL language constructs, the analysis model represented as a Promela program can be of a reasonable size. The largest one in Table 2 is manageable. Moreover, the abstraction technique turns out to be essential for reducing *false negatives*.

Most of the work on formal analysis of BPEL employs a formalism based on the process algebra. H. Foster et al. use FSP and the LTSA model checker[20] for modeling and analysis of BPEL programs.[9] M. Koshkina and F. van Breugel use CCS and Concurrency Workbench (CWB-NC).[16] G. Salaun et al. also use CWB-NC for the behavioral analysis of BPEL.[29]

As for DPE, F. van Breugel and M. Koshkina [3] give an interesting analysis that some execution results may vary depending on whether DPE is introduced or not. They use CCS for the formal modeling and analysis and point out that DPE may introduce unintentional *side effects*.

X. Fu et al. [11] deal with the data-aspect of BPEL as well as the behavioral specification and use the SPIN model checker. They handle XML-based data manipulations by allowing the XPath expression as the guard condition of the guarded automata.

Furthermore, techniques similar to the one for the analysis of BPEL orchestration are employed in choreography languages such as WSCI (Web Service Choreography Interface). A. Brogi et al. work on the formalization and analysis of WSCI.[4] They use CCS for the analysis of the behavioral compatibility. H. Foster et al. use FSP and the LTSA for the problem.[10] Formal analysis of the relation between choreography and orchestration is also an interesting research direction.

7 Conclusion

This paper reports a successful analysis of all four examples in the Business Process Execution Language for Web Services Version 1.1 document.[7] The proposed method takes into account such interesting features as DPE and the abstraction of control variables.

The proposed method and the tool discussed in the paper uses the SPIN as its back-end engine. This method could be combined with the method described in [26] for the analysis of the potential information leaks in BPEL application programs. Since the Web technology is an open-network environment, the security aspect [2] would be an important issue to pursue.

References

[1] T. Ball and S.K. Rajamani, “The SLAM Project: Debugging System Software via Static Analysis,” In *Proc. POPL 2002*, pp.1–3, Jan. 2002.

[2] M. A. Bishop, *Computer Security: Art and Science*, Addison-Wesley, 2003.

[3] F. van Breugel and M. Koshkina, “Does Dead-Path-Elimination have Side Effects?,” Technical Report CS–2003–04, York Univ., Apr. 2003.

[4] A. Brogi, C. Canal, E. Pimentel, and A. Vallecillo, “Formalizing Web Service Choreographies,” In *Proc. WSFM 2004*, pp.73–94, Feb. 2004.

[5] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, “Web Service Description Language(WSDL),” W3C Web Site, 2001.

[6] E. Clarke, O. Grumberg, and D. Peled, *Model Checking*, The MIT Press, 1999.

[7] F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte, and S. Weerawarana, Business Process Execution Language for Web Services, Version 1.1, May 2003.

[8] F. Curbera, R. Khalaf, N. Mukhi, S. Tai, and S. Weerawarana, “The Next Step in Web Services,” *Commun. ACM*, vol. 46, no. 10, pp.29–34, Oct. 2003.

[9] H. Foster, S. Uchitel, J. Magee, and J. Kramer, “Model-based Verification of Web Service Compositions,” In *Proc. ASE 2003*, Sept. 2003.

[10] H. Foster, S. Uchitel, J. Magee, and J. Kramer, “Compatibility Verification for Web Service Choreogrphy,” In *Proc. ICWS’04*, July 2004.

[11] X. Fu, T. Bultan, and J. Su, “Analysis of Interacting BPEL Web Services,” In *Proc. WWW 2004*, pp.621–630, May 2004.

[12] S. Graf and H. Saidi, “Construction of Abstract State Graphs with PVS,” In *Proc. CAV’97*, pp.72 – 83, 1997.

[13] G. J. Holzmann, *The SPIN Model Checker*, Addison-Wesley, 2004.

[14] D. Jackson and J. Wing, “Lightweight Formal Methods,” *In An Invitation to Formal Methods* (*Ed. H. Saiedian*), *IEEE Computer*, vol.29, no.4,pp.16–30, Apr. 1996.

[15] K. Jensen, *Coloured Petri Nets 1*, Springer-Verlag, 1992.

[16] M. Koshkina and F. van Breugel, “Verification of Business Processes forWeb Services,” Technical Report CS–2003–11, York Univ., Oct. 2003.

[17] F. Leymann and W. Altenhuber, “Managing Business Processes as an Information Resource,” *IBM System Journal*, vol.33, no.2, pp.326–348, 1994.

[18] F. Leymann and D. Roller, *Production Workflow: concepts and techniques*, Prentice Hall, 1999.

[19] F. Leymann, Web Services Flow Language (WSFL 1.0), IBM Corporation, May, 2001.

[20] J. Magee and J. Kramer, *Concurrency - State Model & Java Programs*, Wiley, 1999.

[21] M. Mazzara and R. Lucchi, “A Framework for Generic Error Handling in Business Processes,” In *Proc. WSFM 2004*, pp.133–145, Feb. 2004.

[22] L.G. Meredith and S. Bjorg, “Contracts and Types,” *Commun. ACM*, vol. 46, no. 10, pp.41–47, Oct. 2003.

[23] S. Nakajima, "On Verifying Web Service Flows," In *Proc. SAINT 2002 Workshop*, pp.223–224, Jan. 2002.
[24] S. Nakajima, "Verification of Web Service Flows with Model-Checking Techniques," In *Proc. Cyber World 2002*, pp.378–385, IEEE, Nov. 2002.
[25] S. Nakajima, "Model-Checking of Web Service Flow (in Japanese)," In *Trans. IPS Japan*, vol.44, no.3, pp.942–952, Mar. 2003. A concise version presented at OOPSLA 2002 Workshop on Object-Oriented Web Service, Nov. 2002.
[26] S. Nakajima, "Model-Checking of Safety and Security Aspects in Web Service Flows," In *Proc. ICWE'04*, July 2004.
[27] S. Nakajima, "Model-Checking Behavioral Specification of BPEL Applications," In *Proc. WLFM' 05*, July 2005.
[28] S. Narayanan and S. A. Mcllraith, "Simulation, Verification and Automated Composition of Web Services," In *Proc. WWW-11*, 2002.
[29] G. Salaun, L. Bordeaux, and M. Schaerf, "Describing and Reasoning on Web Services using Process Alge bra," In *Proc. ICWS'04*, July 2004.
[30] B. Sarna-Starosta and C. R. Ramakrishnan, "Constraint-Based Model Checking of Data-Independent System," In *ICFEM'03*, 2003.
[31] M.P.Singh and M.N.Huhns, *Service-Oriented Computing*, Wiley, 2005.
[32] S. Thatte, *XLANG - Web Services for Business Process Design*, Microsoft Corporation, May, 2001.
[33] UDDI Web Site, UDDI Technical White Paper. http://www.uddi.org , Sept. 2000.
[34] W. Visser, S. Park, and J. Penix, "Using Predicate Abstraction to Reduce Object-Oriented Programs for Model Checking," In *Proc. FMSP '00*, 2000.
[35] G. Weikum and G. Vossen, *Transactional Information Systems*, Morgan Kaufmann, 2002.
[36] Workflow Management Coalition, "Interface 1: Process Definition Language Process Model," WfMC TC-1016-P (v1.1), Oct. 1999.
[37] P. Wohed, W. van der Aalst, M.Dumas, and A. ter Hofstede, "Pattern Based Analysis of BPEL4WS", Tech. Rep. FIT-TR-2002-04, EUT, 2002.
[38] A. Wombacher, P. Frankhauser, and E. Neuhold, "Transforming BPEL into annotated Deterministic Finite State Automata for Service Discovery," In *Proc. ICWS'04*, July, 2004.
[39] P. Wolper, "Expressing Interesting Properties of Programs in Propositional Temporal Logic," In *Proc. POPL'86*, Jan. 1986.

Shin NAKAJIMA

Professor at National Institute of Infomatics. Received B.S. and M.S. from the University of Tokyo in 1979 and 1981, Ph.D from the University of Tokyo in 2000. Also an adjunct professor at the Graduate University of Avdanced Studies, a SORST reseacher of Japan Science and Technology Agency. Current research areas include formal specification and verification of software, and software modeling techniques.

```
<process name="loanApprovalProcess" targetNamespace="http://acme.com/loanprocessing"
         xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
         xmlns:lns="http://loans.org/wsdl/loan-approval" suppressJoinFailure="yes">
 <partnerLinks>
   <partnerLink name="customer" partnerLinkType="loanPartnerLinkType" myRole="loanService"/>
   <partnerLink name="approver" partnerLinkType="loanApprovalLinkType" partnerRole="approver"/>
   <partnerLink name="assessor" partnerLinkType="riskAssessmentLinkType" partnerRole="assessor"/>
 </partnerLinks>
 <variables>
   <variable name="request" messageType="creditInformationMessage"/>
   <variable name="risk" messageType="riskAssessmentMessage"/>
   <variable name="approval" messageType="approvalMessage"/>
   <variable name="error" messageType="errorMessage"/>
 </variables>
 <faultHandlers>
   <catch faultName="lns:loanProcessFault" faultVariable="error">
     <reply partnerLink="customer" portType="loanServicePT"
            operation="request" variable="error" faultName="unableToHandleRequest"/>
   </catch>
 </faultHandlers>
 <flow>
   <links>
     <link name="receive-to-assess"/>   <link name="receive-to-approval"/>
     <link name="approval-to-reply"/>   <link name="assess-to-setMessage"/>
     <link name="setMessage-to-reply"/> <link name="assess-to-approval"/>
   </links>

   <receive partnerLink="customer" portType="loanServicePT"
            operation="request" variable="request" createInstance="yes">
     <source linkName="receive-to-assess"
             transitionCondition= "getVariableData('request','amount')&lt; 10000"/>
     <source linkName="receive-to-approval"
             transitionCondition= "getVariableData('request','amount')&ge; 10000"/>
   </receive>
   <invoke partnerLink="assessor" portType="lns:riskAssessmentPT"
           operation="check" inputVariable="request" outputVariable="risk">
     <target linkName="receive-to-assess"/>
     <source linkName="assess-to-setMessage" transitionCondition= "getVariableData('risk','level')='low'"/>
     <source linkName="assess-to-approval" transitionCondition= "getVariableData('risk','level')!='low'"/>
   </invoke>
   <assign>
     <target linkName="assess-to-setMessage"/> <source linkName="setMessage-to-reply"/>
     <copy> <from expression="'yes'"/> <to variable="approval" part="accept"/> </copy>
   </assign>
   <invoke partnerLink="approver" portType="loanApprovalPT"
           operation="approve" inputVariable="request" outputVariable="approval">
     <target linkName="receive-to-approval"/> <target linkName="assess-to-approval"/>
     <source linkName="approval-to-reply" />
   </invoke>
   <reply partnerLink="customer" portType="loanServicePT" operation="request" variable="approval">
     <target linkName="setMessage-to-reply"/> <target linkName="approval-to-reply"/>
   </reply>
 </flow>
</process>
```

Appendix A.1 is the BPEL source program for *Loan Approval* taken from the standard document.[7] Its schematic illustration of the process flow is shown in Fig.5.