Available online at www.sciencedirect.com

SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science 151 (2006) 89–105

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Model-Checking Behavioral Specification of BPEL Applications

Shin NAKAJIMA[1]

National Institute of Informatics
and
SORST/Japan Science and Technology Agency
Tokyo, Japan

Abstract

To provide a framework to compose lots of specialised services flexibly, BPEL is proposed to describe Web service flows. Since the Web service flow description is basically a distributed collaboration, writing correct *programs* in BPEL is not easy. Verifying BPEL program prior to its execution is essential. This paper proposes a method to extract the behavioral specification from a BPEL appliation program and to analyze it by using the SPIN model checker. With the adequate abstraction method and support for DPE, the method can analyze all the four example cases in the BPEL standard document.

Keywords: BPEL, Model-Checking, SPIN, Abstraction, DPE.

1 Introduction

Service-oriented computing [15] is an emerging software technology for business networks using the Web technology. Business partners, each acting as a role of a Web service provider, collaborate with each other for customers to benefit from them. As a framework to compose more than one Web services, languages such as WSFL, XLANG, and BPEL4WS are proposed. BPEL4WS (Business Process Execution Language for Web Service), or BPEL for short, is meant to supersede both WSFL and XLANG. And this paper studies BPEL v1.1 [2] being the basis for the OASIS standard.

[1] Email: `nkjm@nii.ac.jp`

doi:10.1016/j.entcs.2005.07.038

Since the Web flow description is basically a distributed collaboration of individual service providers executing concurrently, writing correct *programs* in BPEL is not easy. Concurrency in BPEL is based on a net-oriented work-flow model [8], and faulty behaviors such as deadlocks and violation of properties specific to the application can often sneak in the descriptions. Verifying BPEL program prior to its execution is essential. Actually various methods on the analysis of the behavioral specification are proposed [3][6][7][13][14].

This paper proposes a method to extract behavioral specification from BPEL application program to represent it in a variant of EFA (Extended Finite-state Automaton). The EFA model is then translated into Promela source program and is automatically analyzed by using the SPIN model checker [5]. Although it is possible in principle to translate a BPEL program directly to Promela, using the EFA as the intermediate represenation helps understand what the behavioral specification is and define the verification problem in a concise manner. The proposed method further employs an abstraction of variables having effects on the control aspects of the behavior, and provides an adequate support for DPE (Dead-Path Elimination). It can analyze all the four example cases in the BPEL standard document [2].

2 Formal Analysis of BPEL

2.1 Overview of BPEL

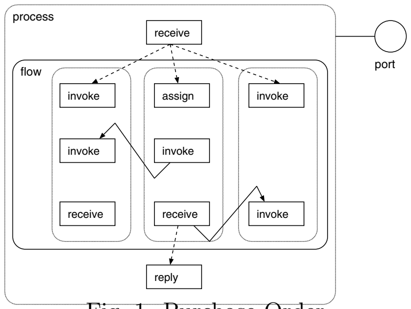

Fig. 1. Purchase Order

BPEL is a behavioral extension of WSDL (Web Service Description Language). The latter is basically an interface description language for Web service providers, which contains information enough for the clients to access. The client invokes a Web service provider using WSDL. The invocation is *one-shot*; WSDL does not describe global states of the provider.

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.

Figure 1 illustrates a typical BPEL process, Purchase Order example in [2]. The concurrency in BPEL is based on the idea of the net-oriented concurrent computation model in that a flow graph, described by `flow` activity and control links (`link`), represent concurrency. The example BPEL program first waits for an invocation requests from the outside with the `receive` activity, and then initiates three concurrent activities enclosed in the `flow`. After all the concurrent executions terminate, the control goes to the `reply` activity which returns some value to the original outside initiator as the result of the computation process. The schematic diagram also shows by using the solid lines (`Link`'s) that some control dependencies exist between some of the basic activities.

BPEL, as its full name suggests, is characterized as a business process description language in the Web service architecture, and shares many of features with the work-flow schema languages [16]. The concurrent aspect of BPEL (`flow` activity) inherits from WSFL, which in turn borrows its core ideas from PM-Graph [8], that is a model of work-flow systems.

2.2 Two Aspects of Formal Analysis

A BPEL process exchanges messages with the partner Web service providers, and each message confirms to a particular WSDL message type. In BPEL, `<define>` tag introduces the definitions for the messages, and `<process>` tag defines how the message data are manipulated and how the control flow proceeds, i.e. the behavioral specification. The correctness of BPEL programs also has two kinds of distinctive characteristics: message type conformance and behavioral specification.

The message type conformance is a check to see whether the type of a exchanged message does not contradict with the message type expected by the port through which the message is sent or received. And the problem can be considered as a variant of type-checking seen in typed programming languages. This paper does not go further into this direction.

Alternatively, this paper focuses on the behavioral aspect of the system. It is because the behavioral specification plays a crucial role in distributed concurrent systems such as BPEL programs. Concurrent systems may sometimes have potential deadlocks. Such a faulty behavior is not easy to uncover once a system starts its executions because it occurs non-deterministically. The system shows such a faulty behavior only in some situations, and appears to execute correctly in most of the times.

2.3 Related Work

The earliest work on the formal verification of Web service flow can be traced back to the paper by S. Narayanan and S.A. Mcllraith who employs Petri-net for the automatic analysis [13]. They, however, do not study the languages relating to the Web service standard.

First proposals that aim to the standard technology are WSFL and XLANG announced in May 2001. And S. Nakajima mentions a basic idea to use the model-checking techniques for the analysis of WSFL [9][10]. Although WSFL is obsolete, the basic technique to deal with concurrency including DPE is still applicable to BPEL. And the detailed technique to use the SPIN model-checker is found in [11]. This paper employs the same technique to handle DPE for the case of BPEL, and studies what techniques would be needed to analyze the BPEL application programs by using the model-checker. And the abstraction technique is turned out essential.

Most of the work for the formal analysis of BPEL employs the formalism based on the process algebra. H. Foster et al use FSP and the LTSA model-checker for modeling and analysis of BPEL programs [3]. M. Koshkina and F. van Breugel use CCS and Concurrency Workbench (CWB-NC) in their work[7]. G. Salaun et al also use CWB-NC for the behavioral analysis of BPEL [14]. As for DPE, F. van Breugel and M. Koshkina [1] give an interesting analysis that execution results may be different depending on whether DPE is introduced or not, leaving un-intentional *side-effects*. Last, X. Fu et al deal with the data-aspect of BPEL by using the abstraction technique as well as the behavioral specification and use the SPIN model-checker [6]. The abstraction technique, however, is not always sound.

3 Behavioral Specification in BPEL

3.1 BPEL Behavioral Specification

3.1.1 Basic Process Description

Basic computation elements in BPEL are activities to represent atomic actions. BPEL provides three kinds of activities to exchange information with the 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, and waits for the result.

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

In addition to the communication primitive activities, BPEL provides an `assign` activity for accessing variables. It also has other activities concerning

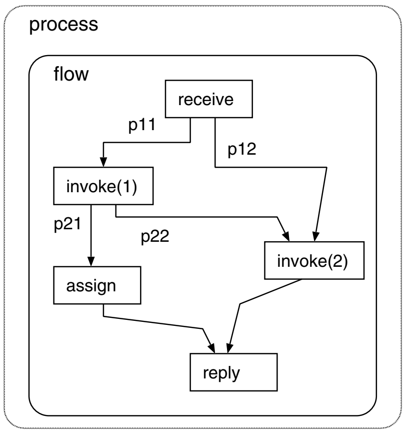

Fig. 2. Loan Approval

to implement control flows such as `sequence` (sequential executions), `switch` (branch on conditions), `while` (repetitions), and `flow` (concurrency).

BPEL introduces a lexical context with `scope` activity. The lexical context defines an effective scope of variables and various handlers such as exception. However, in view of determining control flows, a serializable scope is important. A `scope` activity can have a `serializable` attribute, which specifies multiple concurrent accesses to the shared resources are serialized. And the semantics is very similar to the standard isolation level *serializable* used in the database transaction [2]. It is defined in terms of *strict two-phase locking* protocol.

3.1.2 Concurrency in BPEL

The `flow` activity introduces a flow graph, consisting of the `activity`'s as nodes and `link`'s as edges representing control links. This paper adapts the semantics from the book [8] [2], that is based on the PM-flow proposed as a work-flow schema language. The operational semantics of BPEL is essentially a set of rules to select activities to fire.

Figure 2 is an example from [2], `Loan Approval`, to use a `flow` activity and `link`'s. The `flow` has five atomic activities, that are executed concurrently, but have causal dependencies specified with the `link`'s. The `receive` has two outgoing control links, each of which is set *true* depending on the values of some other variables. Figure 2 uses variables such as `p11` and `p12` to denote the condition schematically. The `invoke(1)` on its left downward similarly has two links and their values determined by `p21` and `p22`. Two variables are

[2] According to an informal conversation with the author of [2], their implementation of BPEL system has been adapted the semantics in [8].

introduced because the two links are distinctive in BPEL text, although they are logically related.

Once the execution control is passed to the `flow` activity, its inside activities start their execution concurrently. The `receive`, among others, can be fired since no condition is posed on it. It sets the values of its outgoing links depending on the values of some variables. Although Figure 2 compactly illustrates that the condition is `p11`, `p11` refers to the `transitionCondition` (`tC` for short) attribute of the following code fragment.

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

When `p11` is *true*, the `invoke(1)` is then chosen and the values of its two outgoing links are determined similarly. Another variable `p12` is supposed to be *false* when `p11` is *true*. Further, when `p21` becomes *true*, the `assign` is selected and the control finally goes to the `reply` at the bottom.

3.1.3 Dead-Path Elimination

The operational semantics of the flow is somewhat complicated due to DPE (Dead-Path Elimination). Figure 3, extracted from the book [8], explains why DPE is needed.

In Figure 3, the activity `A` has its result `p` to be *true* and the `r` of the activity `B` *false*. Since `r` is *false*, the activity `C` will never be executed. Therefore, the join condition of the activity `D` will also never be evaluated. The 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, the activity `D` can be executed logically, but its join condition will never be evaluated in the naive semantics.

The DPE provides a means to resolve such pseudo faulty situations. DPE starts its execution when a join condition becomes *false*, or when there is an activity having a single input control 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. DPE uses *false* values when it involves logical calculation. DPE terminates the propagation process when it reaches either an activity having a join condition or an ultimate end.

The example in Figure 2 requires DPE for executing correctly. When both `p12` and `p22` turn out to be *false*, the join condition of the `invoke(2)` activity becomes *false* and the activity is never executed. Then 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 the `invoke(2)` activity. On the other hand, since the outgoing link from the `assign` activity becomes *true*, the `reply` activity will logically be executable.

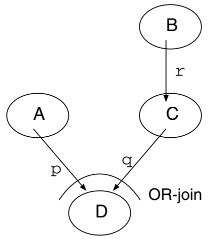

Fig. 3. Dead-Path Elimination

3.2 Formal Analysis and Abstraction

Abstraction is essential for the analysis of the behavioral specification. Since the analysis is done in the 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 effect on the control flow of the BPEL program to be analyzed. Further, the analysis is a process to perform without knowing actual values. All the values that might potentially be possible would be checked in the analysis. It, however, is not feasible in most cases because the number of combinations of all the values would be huge.

As explained for the example in Figure 2, the `tC` attribute in `<source>` tag should be appropriately evaluated in order to have valid executions. The example condition involves a value stored in the variable `request`, which is assigned in the `receive` activity. Since the actual value is not determined at the analysis time, the best to say is that the `tC` would be either $true$ or $false$ in an equal probability. In other word, the link takes either value in a non-deterministic manner.

If such non-determinism is applied to all the `tC`'s independently, the two outgoing links of the `receive` activity can be either $true$ or $false$, and both take $true$ at the same time in some cases. For example, it is the case, in Figure 2, that both `p11` and `p12` take $true$, which is not what is meant in the original BPEL application. Therefore, in order that the flow executes correctly, the two outgoing links should take distinctive values. Namely, when `p11` is $true$, `p12` should be $false$.

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

As for the example in Figure 2, two predicate variables, $pred1$ and $pred2$, are introduced to represent the conditions on the two outgoing links from the `receive` activity.

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

where `request.amount` is an abbreviation of

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

And the two definitions has the logical relationship of $pred2 = \neg\ pred1$.

Further, although the value of the BPEL variable `request` is assigned in the `receive` activity, the actual value is not determined at the analysis time. Thus, `receive` activity should be considered as being accompanied with an assignment that $pred1$ takes either $true$ or $false$ in a non-deterministic manner. Another predicate variable $pred2$ is appropriately assigned by consulting the above logical relationship.

Introducing predicate variable is an idea relating to the *predicate abstraction* [4]. And the approach in this paper is meant to be structure-preserving. Namely, the description after the abstraction has the same structure of the original BPEL program.

4 Modeling and Analysis with EFA

This paper adapts the automaton-theoretic techniques for the behavioral analysis of BPEL programs. The intermediate representation is meant to introduce for the two purposes: (1) to define what the behavioral specification of BPEL programs is, and (2) to clearly state the verification problem at hand.

4.1 Modeling with EFA

4.1.1 Extended Finite-state Automaton

Behavioral specification is essentially an abstract view of the system in terms of the control flow, and thus automaton is a good tool for the representation and analysis. BPEL, however, has language constructs relating to the data flow aspects: some activities exchange messages with the partner service providers via partner links, and the incoming messages are stored in the variables. Further, variables and links may have effect on the control flow: variables may appear in expressions of the condition in `switch` and `while`, and also may be used in the condition to fire particular links in `<source>` tag. 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 by 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 $\mathcal{A}$. It is also a triple consisting of an alphabet (Σ), a variable updating function (θ) and a guard ($\mathcal{G}$). 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 on the condition that the accompanied guard condition is *true*. And the variable updating function is executed in the course of the transition, which assign a new value to the specified variable in the target location point. A variable updating function is actually a set of simultaneour assignments.

Each alphabet in Σ may take one of the three forms. The two forms, P ! X and P ? X, are in relation to communication with the external environment or automaton, while ϵ is an internal action. Specifically, P ! X is an output action designator while P ? X is an input one. Operationally, the output action is a message send that the value X is sent to the communication channel P. The input action is a message receive that a message coming from the channel P is received and its value is set to the variable X. Last, an asynchronous product of two EFA's is defined in a standard way.

4.1.2 EFA for BPEL

In order to take into account the features specific to BPEL language constructs, below introduces a customized version of EFA, M_{BPEL}. First, the communication channels appeared in the input/output designators represent `partnerLink`'s 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

$\mathcal{G}_{PL}$: Guard Condition

$\mathcal{V}_B$ denotes a set of variables appeared explicitly in the source BPEL program. They are extracted from `<variables>` tag in `<process>` description. $\mathcal{V}_L$ is a set of link variables, each of which corresponds to a `<link>` introduced in `<links>` tag of `<flow>` activity. The `<link>` is employed to specify control flow among the concurrently executing activities in a `<flow>`, and each link can be regarded as a boolean variable. It is set *true* when the control flow does exist. $\mathcal{V}_P$ 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 appeared in `switch`, `while`, or `tC` of `source` tag. For the example in Section 3.2, *pred*1 and *pred*2 are the predicate variables.

Variables in either $\mathcal{V}_L$ or $\mathcal{V}_P$ are boolean-valued, but BPEL variables in $\mathcal{V}_B$ are application-specific and may take values in an infinite domain. When the domain is infinite, or is huge if not infinite, taking into account all the potential values in the analysis process is not feasible. This paper assumes that a BPEL variable in $\mathcal{V}_B$ takes a value from a finite set consisting only of the two elements, $definite$ and $undefined$. This implies that the analysis on the data apsect is very limited; data value is ignored and how a data token is flowed down along the data-flow is a concern here.

The guard condition is 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 $\mathcal{V}_B$. The guard condition expression g is either a simple boolean-valued expression or equality (non-equality) of two. The EFA for BPEL is data-independent [17].

4.2 Verifications

The analysis technique is based on a state explosion search for verifying behavioral specification of BPEL programs, actually expressed in terms of EFA's. Some definitions are necessary to define the verification problem.

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

$$\sigma^{\omega} = q_0 q_1 q_2 \dots$$

where $\forall q_i \in \mathcal{Q}$, and the stutter extension rule is assumed to represent a finite run as an infinite one. And an accepting run is also defined by following a standard definition as

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

An accepting run is an successful execution path of the BPEL program.

The first verification problem is concerned with the *reachability* and often refereed to as a check for the deadlock freedom. It is a test whether a BPEL program does not stop its execution in an accidental manner, which can be stated as a test whether there is a run that is not an accepting one.

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

LTL formula takes an atomic proposition, which is a boolean expression to be formed by using information at each location point. The expression may refer to values of BPEL variables since the EFA has a variable map ρ_B in its definition making it possible to obtain BPEL variables in each location point.

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

4.3 BPEL to EFA

This section presents an overview of the translation scheme from BPEL activity to EFA fragment. The translations of atomic activities are mostly straight forward. Since EFA already has the notions of send-receive 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. Since the `invoke` activity is roughly said to be a combination of a message send and receive, the EFA fragment consists of two consecutive transitions.

Complex activities forming control sequences may make use of the guard condition of EFA. The `switch` activity is a multi-way conditional branching control and has a `otherwise` branch. The `while` activity introduces an iteration control and requires a loop structure to represent the repetition.

The translation of a `flow` activity consists of two steps. First, the top `flow` activity 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 *sub* EFA's executing concurrently, and waits for their completions. Second, all the EFA's are composed to be an EFA by taking their asynchronous product.

In order to synchronize the executions, the control variables are introduced. For each *sub* EFA, two boolean variables `startX` and `endX` are implicitly introduced in $\mathcal{V}_L$. The `startX` is set to *true* when the execution control is entered into the `flow`, and thus the *sub* EFA starts its execution since the guard condition on the transition becomes *true*. Upon its completion, the *sub* EFA sets the `endX` variables *true*. The *main* ensures that all the sub-activities are terminated by using the appropriate guard condition.

An atomic activity enclosed in `flow` may have synchronization in regard to its incoming links. The enclosed activity has a particular synchronization condition as its `joinCondition` attribute. The condition is represented as a guard condition using the appropriate link variables.

An activity may also have `<source>` tags with `tC`. The `<source>` tag sets its attribute `linkName` *true* when the `tC` is *true*, and $false$ otherwise. It means that the `linkName` is always set to the value of `tC`, and thus it can be taken care of with the variable update function.

5 Implementation with SPIN

5.1 EFA to Promela

This section presents a method to use the SPIN model-checker for the representation of the EFA model and the analysis. The basic idea is just to translate an EFA into a Promela source program. Promela is the specification language for the SPIN, and adapts a computational model of communicating finite-state automaton 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 an automaton.

- An EFA automaton M becomes a Promela process.
- A communication channel P appeared in the input and output action designators is translated to a Promela channel. Since the channel P denotes `partnerLink` in BPEL, the name of the Promela channel *name* is taken from the `name` attribute. And the Promela channel declaration takes into account the *type* of the message exchanged.

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

  where `mtype` is a enumeration type describing the `operation (Op)` and the second argument carries the data value, actually a data token.
- Variable $\mathcal{V}$ is translated to a (global) variable in Promela. Specifically, predicate variables $\mathcal{V}_P$ are boolean Promela variables initialized to be $false$.

```
bool name = false;
```

Link variables $\mathcal{V}_L$ are also basically boolean, but a slight different encoding is used in order to deal with DPE.

- Encoding 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 by a Promela sequencing (`;`). Conditional branching in EFA, making use of the guard condition $\mathcal{G}$, uses Promela multi-way branch (`if...fi`). Repetition is easy to represent in Promela since the language provides the looping construct (`do...od`).
- Input/Output action designators are Promela channel operations. Both P ? X and P ! X are translated to Promela counter-parts. The channel P is defined as a Promela channel as discussed above.

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

- A variable update function θ denotes a set of simultaneous assignment of multiple variables. In its Promela translation, the atomicity (`atomic{...}`) is introduced.

In order to analyze the behavior, the Promela model should be *closed*. A closed model consists of the Promela process 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.

5.1.2 Abstraction and Static Analysis

As discussed in Section 3.2, abstraction is necessary to obtain an EFA from a BPEL program. It requires a static analysis to introduce appropriate predicate variables ($\mathcal{V}_P$). As briefly discussed with a simple example, the analysis is basically a define-use chain (du-chain) of variables having effects on the control flow. Further, a single predicate variable is introduced for each conditional expression.

Although the example in Figure 2 is simple, the translation of the `receive` activity might be interesting. It requires two predicate variables logically related. The fragment relevant to the discussion here follows.

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

And the Promela code fragment looks as below.

```
Customer ? Request(request) ;              /* receive */
if :: pred1 = true  :: pred1 = false fi ; /* non-determinstic */
pred2 = ! pred1 ;                          /* logical relation */
receiveToAssess = pred1 ;                  /* source */
receiveToApproval = pred2 ;                /* source */
```

The variable `pred1` denotes to the expression `request.amount < 10000`, which depends on the variable `request`. The d-location of `pred1` is at the `receive` activity where it is assigned a new value to the variable `request`. Its u-location is at the first `<source>` tag, in which the value of `pred1` is accessed to determine the value of the link. Although it is simple, it can be seen that a du-chain analysis calculates a correct causal relationship between the two locations.

The next thing is to determine the value. However, the best to say is that the variable `pred1` would be either `true` or `false` because the variable `request` is not determined at the analysis time. Above is inserted the code fragment to set the variable `pred1` non-deterministically either one. Once the value of `pred1` is assigned, the search algorithm used in the model-checking method of SPIN can explore all the possible combination of the values.

5.1.3 DPE and Serializable Scope

Most of the translation is presented above, however, some special care should be taken in regard to the DPE and the `<scope>` tag with `serializable` flag *true*.

In order to handle DPE properly, the set that the link variable takes as its value extends to have *forced*. The *forced* value is generated and flowed 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 coming *forced* value makes the activity not executed but is further flowed down along the outgoing link(s) of the activity. Some details can be found elsewhere [11].

The serializable scope specifies multiple concurrent accesses to the variables inside the scope are serialized. The serializable scope should be considered to form a critical region. The Promela version 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 the shared variables looks as below.

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

Table 1
Four Cases

	Name	BPEL Features	#States
(1)	Purchase Order	variable, flow	249
(2)	Shipping Service	switch, while	21
(3)	Loan Approval	flow, DPE	3516
(4)	Auction Service	multiple start	57

5.2 Example Cases

Table 1 presents a summary of the experiment. The four cases are taken from the BPEL standard document [2]. The last column shows the number of states in the analysis using the SPIN model-checker. Although its exact value is not significant, the number shows roughly how the analysis is complicated. As discussed in Section 5.1, the translation makes fully use of Promela language constructs, the state-space can be small in the most example. The third example Loan Approval contains five activities executing concurrently, the state-space becomes large due to the interleaving semantics of the concurrency.

Each example employs particular language features in BPEL. (1) Purchase Order (Figure 1) is introduced as the initial example to illustrate the most basic structures and some of the fundamental concepts of BPEL language. (2) Shipping Service uses `switch` and `while` activities to implement adequate controls with some variables. Since their values are not determined, abstraction is essential. (3) Loan Approval (Figure 2) makes use of BPEL concurrency with the `flow` activity. This example also needs the abstraction of variables having effects on the evaluation of the transition conditions. (4) Auction Service is an example to have multiple start activities. It adds no new features for the analysis in this paper, but it is interesting to note that a manual translation can reduce the size of the state-space about 25% smaller. In each case, an appropriate *environment* Promela description is introduced. The environment process is constructed manually so that it adds little effects on the size of the state-space.

As discussed in Section 4.2, LTL is a handly tool to express certain application-specific properties. For the third example Loan Approval (Figure 2), a property that either `assign` or `invoke(2)`, not both, is executed can be checked by expressing as below.

$$\texttt{[]}(\text{receive} \rightarrow (\texttt{<>}\text{assign} \wedge \texttt{[]}\neg\ \text{invoke}(2) \vee \texttt{<>}\text{invoke}(2) \wedge \texttt{[]}\neg\ \text{assign}))$$

And the SPIN model-checker ensures that the property holds.

6 Discussion and Conclusion

This paper is the report of the first successful results that can analyze all the four examples in the document [2]. The key point is that the proposed method takes into account of such interesting features as DPE and the abstraction of control variables.

Thanks to using the EFA as the intermediate representation, the proposed method is clearly understood in two points: (1) to define what the behavioral specification of BPEL programs is, and (2) to clearly state the verification problem at hand. Further, X. Fu et al [6] employs the guarded automaton model, which is essentially the same as the EFA in this paper. Both can use variables and the transition may have guard condition. Such extended automata is a good to for the behavioral analysis of BPEL application programs.

Other related work on the model-checking of the BPEL programs do not consider DPE, and cannot properly analyze the Loan Approval example. In addition, most of them do not care much about the abstraction on the control variables, and do not introduce techniques such as the predicate abstraction. The analysis results may have more *false negatives* than the approach in this paper. Such a *false negative* sometimes appears as a result of the over-approximation ignoring the causal relationships between the variables having effects on the execution control.

Unfortunately not all the BPEL programs can be analyzed with the proposed method. It does not deal with the semantics of handlers such as exception or compensation. Since handlers are the important language constructs having much effects on the behavioral specification of BPEL programs, incorporating such features in the analysis method is essential. It is one of the futre work.

Last, the tool discussed in the paper uses the SPIN as its back-end engine, and will be combined with the method in [12] for the analysis of the potential information leakage in BPEL application programs.

References

[1] F. van Breugel and M. Koshkina. Does Dead-Path-Elimination have Side Effects? Technical Report CS-2003-04, York Univ., April 2003.

[2] F. Curbera et al. Business Process Execution Language for Web Services. Version 1.1, May 2003.

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

[4] S. Graf and H. Saidi. Construction of Abstract State Graphs with PVS. In *Proc. CAV'97*, pages 72 – 83, 1997.

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

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

[7] M. Koshkina and F. van Breugel. Verification of Business Processes for Web Services. Technical Report CS-2003-11, York Univ., October 2003.

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

[9] S. Nakajima. On Verifying Web Service Flows. In *Proc. SAINT 2002 Workshop*, pages 223–224, January 2002.

[10] S. Nakajima. Verification of Web Service Flows with Model-Checking Techniques. In *Proc. Cyber World 2002*, pages 378–385, IEEE, November 2002.

[11] S. Nakajima. Model-Checking of Web Service Flow (in Japanese). In *Trans. IPS Japan*, Vol.44, No.3, pages 942–952, March 2003. A concise version presented at OOPSLA 2002 Workshop on Object-Oriented Web Service, November 2002.

[12] S. Nakajima. Model-Checking of Safety and Security Aspects in Web Service Flows. In *Proc. ICWE'04*, July 2004.

[13] S. Narayanan and S.A. Mcllraith. Simulation, Verification and Automated Composition of Web Services. In *Proc. WWW-11*, 2002.

[14] G. Salaun, L. Bordeaux, and M. Schaerf. Describing and Reasoning on Web Services using Process Algebra. In *Proc. ICWS'04*, July 2004.

[15] M.P.Singh and M.N.Huhns. *Service-Oriented Computing.* Wiley 2005.

[16] P. Wohed, W. van der Aalst, M.Dumas, and A. ter Hofstede. Pattern Based Analysis of BPEL4WS. Techinical Report FIT-TR-2002-04, EUT, 2002.

[17] P. Wolper. Expressing Interesting Properties of Programs in Propositional Temporal Logic. In *Proc. POPL'86*, Janurary 1986.